ARTIST

FP7-317859

Advanced software-based seRvice provisioning and migraTIon of legacy Software

---

Deliverable D9.5

Model-to-Code Transformations for Specific Cloud Infrastructures

---

Editor(s):	Alexander Bergmayr, TUWIEN Javier Troya, TUWIEN Manuel Wimmer, TUWIEN
Responsible Partner:	TUWIEN
Status-Version:	Final – v1.0
Date:	30/09/2014
Distribution level (CO, PU):	PU

Project Number:	FP7-317859
Project Title:	ARTIST

Title of Deliverable:	Model-to-code transformations for specific cloud infrastructures
Due Date of Delivery to the EC:	30/09/2014

Workpackage responsible for the Deliverable:	WP9 - New software generation by forward engineering
Editor(s):	Alexander Bergmayr, TUWIEN Javier Troya, TUWIEN Manuel Wimmer, TUWIEN
Contributor(s):	Juncal Alonso, Tecnalia Girja Echevarria, Tecnalia Aliki Kopaneli, ICCS Jesús Gorroñogoitia, ATOS Konrad Wieland, Sparx
Reviewer(s):	Oliver Strauß, Fraunhofer
Approved by:	All Partners
Recommended/mandatory readers:	WP6, WP9, WP10, WP11

Abstract:	This deliverable comprises a set of code generation templates which are tailored to specific Cloud infrastructures
Keyword List:	Forward Engineering, Model Transformation, Software Migration
Licensing information:	Generally EPL (open source), indicated otherwise. The document itself is delivered as a description for the European Commission about the released software, so it is not public.

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
v0.1	25/06/14	Initial Table of Contents	TUWIEN
v0.2	22/07/14	Refactoring of Table of Contents	TUWIEN
v0.3	08/09/14	Integration of partner contributions	TUWIEN (ATOS, ICCS, Sparx, Tecnalia)
v0.4	12/09/14	Revision of partner contributions	TUWIEN (ATOS, ICCS, Sparx, Tecnalia)
v0.5	15/09/14	Addition of Introduction, Conclusions, and Executive Summary	TUWIEN
v1.0	26/09/14	Final version	TUWIEN

Table of Contents

Table of Contents 4
Table of Figures 6
Table of Tables 6
Terms and abbreviations 7
Executive Summary 9
1 Introduction 11
1.1 About this deliverable 11
1.2 Fitting into overall ARTIST solution 12
1.3 Innovations 12
1.4 Document structure 13
2 Code Generation Considerations 14
2.1 Toolbox Global Architecture 14
2.2 Code Generation Frameworks – State of the Art 15
2.3 Traceability from Model to Code and Back Again – Reuse of Existing Code 18
2.4 User Interaction: Code Generator Configurations and Extensions 19
3 Implementation 22
3.1 Functional description 22
3.2 UML2Java Code Generator – Technical Description 22
3.2.1 Prototype architecture 23
3.2.2 Components description 24
3.2.2.1 Generic Code Generator 24
3.2.2.2 Code Generator Configuration 25
3.2.2.3 Code Generator Extensions 25
3.2.3 Technical specifications 25
3.2.4 User Manual 25
3.3 UML2C# Code Generator – Technical Description 28
3.3.1 Prototype architecture 30
3.3.2 Components description 30
3.3.3 Technical specifications 32
3.3.4 User Manual 32
4 Evaluation 36
4.1 Completeness and Correctness of Generated Code 36
4.1.1 Completeness 36

4.1.2 Correctness 36
4.1.3 User Manual 42
4.2 Maintainability/Understandability of Generated Code 43
4.2.1 Technical description 44
4.2.1.1 Prototype architecture 44
4.2.1.2 Components description 44
4.2.1.3 Technical specifications 45
4.2.2 Delivery and usage 45
4.2.2.1 Package information 45
4.2.2.2 Installation instructions 45
4.2.2.3 User Manual 45
4.2.2.4 Licensing information 48
4.2.2.5 Download 48
5 Delivery and Usage 49
5.1 Package Information 49
5.2 Installation Instructions 49
5.3 User Manual 49
5.4 Licensing Information 49
5.5 Download Instructions 49
6 Conclusions 50
7 References 51
APPENDIX: Core elements of the UML Class Diagram Language 53

Table of Figures

FIGURE 1: TRANSITION FROM PSM TO CODE 11
FIGURE 2 TRACEABILITY IN REVERSE ENGINEERING WITH RTT 18
FIGURE 3 TRACEABILITY IN FORWARD ENGINEERING WITH RTT 19
FIGURE 4 CONFIGURATION OF GLOBAL SETTINGS FOR THE CODE GENERATOR 20
FIGURE 5 OPTIONS FOR THE CODE GENERATOR 20
FIGURE 6 PROGRAMMING LANGUAGE SPECIFIC PARAMETERS FOR THE CODE GENERATOR 21
FIGURE 7 ARCHITECTURAL OVERVIEW OF THE UML2JAVA CODE GENERATION FACILITY 23
FIGURE 8 OBJECTIFIED DOMAIN MODEL (PSM) 26
FIGURE 9 GENERATED SERVICE CLASSES (PSM) 26
FIGURE 10 RUN CONFIGURATION OF *UML2JAVA* CODE GENERATOR 27
FIGURE 11 C# VALIDATION PROCESS 31
FIGURE 12 C# CODE GENERATION PROCESS 32
FIGURE 13 VERIFYING CORRECT INSTALLATION OF ARTIST EA ADD-IN 32
FIGURE 14 ACTIVATING AND DEACTIVATING VALIDATION RULES IN THE C# CODE GENERATOR 33
FIGURE 15 DETECTION OF ERRORS IN THE MODEL VALIDATION RESULTS 33
FIGURE 16 METRICS PROVIDED IN THE MODEL VALIDATION RESULTS 34
FIGURE 17 DIALOG TO GENERATE CODE 35
FIGURE 18 – UNIDIRECTIONAL ASSOCIATIONS 37
FIGURE 19 – REPRESENTATION OF QUALIFIERS 38
FIGURE 20 – SOURCE END MULTIPLICITY 39
FIGURE 21 – BI-DIRECTIONAL ASSOCIATIONS 39
FIGURE 22 – REPRESENTATION OF BI-DIRECTIONAL ASSOCIATION 39
FIGURE 23 – REPRESENTATION OF INHERITING ENUMERATIONS 40
FIGURE 24 EXECUTING THE ACCELEO (OBEO) CODE GENERATOR ON A TEST CASE 43
FIGURE 25 – MMC HIGH LEVEL ARCHITECTURE 44
FIGURE 26 PACKAGE STRUCTURE OF THE MAINTAINABILITY METRIC CALCULATOR 46
FIGURE 27 MAINTAINABILITY METRIC CALCULATOR PROJECT 46
FIGURE 28 TESTING PACKAGE 47
FIGURE 29 MAINTAINABILITY METRIC CALCULATOR TESTING 47

Table of Tables

TABLE 1 MAPPING BETWEEN UML CONCEPTS AND JAVA CONCEPTS 24

Terms and abbreviations

API	Application Programming Interface
CD	Class Diagram
CDN	Content Delivery Network
CGF	Code Generation Framework
CI	Computer Independent
CTF	Code Template Framework
CIM	Computing Independent Model
CRUD	Create, Read, Update, Delete
DNS	Domain Name Service
EA	Enterprise Architect
EASL	Enterprise Architect Simulation Library
EC	European Commission
EMF	Eclipse Modeling Framework
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
J2EE	Java 2 Enterprise Edition
JSON	JavaScript Object Notation
JSP	JavaServer Pages
M2M	Model-to-Model
M2T	Model-to-Text
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MMC	Maintainability Metric Calculator

MOF	Meta-Object Facility
MTL	Model-to-Text Language
OCL	Object Constraint Language
OMG	Object Management Group
OOPL	Object-Oriented Programming Language
PaaS	Platform as a Service
PDM	Platform-Deployment Model
PIM	Platform-Independent Model
PSM	Platform-Specific Model
RCP	Rich-Client Platform
REST	Representational State Transfer
RTT	Reusability Trace Tool
SDK	Software Development Kit
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

Executive Summary

The migration phase of the ARTIST project consists of two steps. The first one, namely reverse engineering step, deals with the abstraction of the original application into platform-independent models (PIMs), which keep no information about any specific platform. The second step, forward engineering, has as final target the production of code targeted at a specific platform. In the process of obtaining this final code, the PIMs are first transformed into platform-specific models (PSMs), which contain information about the particular platform where the application is to be eventually deployed. After the PSMs have been obtained, the work presented in this deliverable can be applied, namely the generation of code from models.

This document describes the code generation approach implemented as model-to-text transformations developed in the context of the ARTIST project until M24. In particular, this document describes the design rationale behind the code generators. One of the main design rationales is to provide reusable code generators for the core of UML, which is considered to be the UML Class Diagram. The code generators are designed to be extensible for different cloud target platforms in order to counteract the recreation of standard code generation functionality that is needed for targeting general programming languages, i.e., in the context of ARTIST, Java and C#.

One may assume that available code generators are providing sufficient support for the UML class diagram modeling concepts as there is a plethora of different UML-based code generators available in open-source as well as in commercial tools. However, after evaluating several code generators, we explored that only some elements are properly treated by the available code generators and that substantive extensions of those are needed in order to cover a broader spectrum of the UML Class Diagram language. In this respect, we have in particular focussed on the constraints imposed by UML Class diagrams for their Object Diagrams, i.e., their instances, to be reflected as well in the generated code. Especially, the aspect that a UML Class Diagram also represents a constraint language for the object structure is mostly neglected by current code generators as they mostly consider the possibility to instantiate objects from classes and setting their values, but in a more or less unrestricted way.

This document therefore explains several approaches for implementing model-to-text transformations in order to realize code generators as well as important aspects such as how code generators may be extended or configured. Subsequently we explain two of our current code generators. While the first one is dedicated to Java, based on an open source code generator, and customized for one particular cloud platform, the second one is an extension of a code generator currently available in the Enterprise Architect tool considering C# code generation. Both generators come with dedicated contracts which act as pre-conditions to check the validity of the input models. Here not only the constraints of the UML metamodel are considered, but also the target constraints of the output languages such as naming rules and reserved keywords are checked before the code generation is actually started. By this additional step, many errors such as compile time errors in the generated code can be prevented.

As we have realized that developing code generators may sound simpler than it actually is, we report also on the validation of code generators by reasoning about completeness and correctness of the generators as well as the non-functional properties of the generated code such as readability and maintainability. For this validation concerns, we present a dedicated methodology which could be used also in future generator developments.

By contributing first versions of base code generators for Java and C#, by demonstrating how these code generators can be extended and configured for particular cloud platforms, and by showing how they can be validated, we pave the way to develop further cloud provider specific code generators in the last year of the ARTIST project which will be reported in the context of WP 10 in M30.

1 Introduction

After having obtained Platform-Specific Models (PSMs) from Platform-Independent Models (PIMs) and Platform-Deployment Models (PDMs) [1], the next step in the forward engineering process is to obtain code from the obtained PSMs. These PSMs have several (UML) profiles applied that represent different cloudification concerns. The model-to-text (M2T) transformations have to access both the UML concepts as well as those from the stereotypes used and convert them into different programming languages. In the ARTIST project, we focus on the Java and C# code generation (cf. Figure 1).

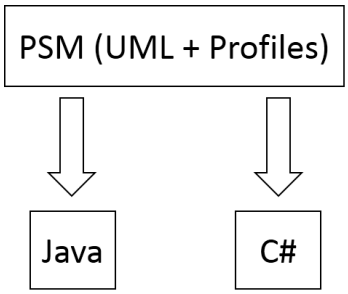

Figure 1: Transition from PSM to code

1.1 About this deliverable

By following the MDA proposal[1], transformations should be used to automate the transition from PSMs to code. Without a model-driven approach or in model-based approaches, this transition is done manually, i.e., the models are the blueprints to develop the systems by hand. Thus, if transformations are considered to speed-up and systematize this transition, they need to have the knowledge of which concepts of programming languages relate to which concepts in modelling languages. In this deliverable, we have targeted the production of code from PSMs starting from existing UML to code generators. The reason for the need to develop new code generators, though, or extend existing ones, is that, despite the extensive number of research works, papers and frameworks that exist targeting the generation of code from models [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], there are still many limitations and the need to improve these approaches. Furthermore, there is still the need to obtain a unified version, since different approaches propose different solutions.

By studying the different approaches for the generation of code, one soon realizes that there are still many gaps in the mapping from UML to object oriented languages. This is because the semantics of UML are not clearly defined, and different developers may interpret them in different ways. Furthermore, the different levels of abstraction between modelling languages and programming languages, together with the different features and ways of trying to achieve the same behaviour (think, for instance, of the behaviour of associations both in UML and Java, which is related to the semantics), complicates the mapping.

One of the main design rationales is to provide reusable code generators for the core of UML, which is considered to be the UML Class Diagram. The code generators are designed to be extensible for different cloud target platforms in order to counteract the recreation of standard code generation functionality that is needed for targeting general programming languages, i.e., in the context of ARTIST Java and C#. In this deliverable, we particularly focus on the constraints imposed by UML Class diagrams for their Object Diagrams, i.e., their instances, to be reflected as well in the generated code. Especially, the aspect that a UML Class

[1] http://www.omg.org/mda/

Diagram also represents a constraint language for the target object structure is mostly neglected by current code generators as they mostly consider the possibility to instantiate objects from classes and setting their values, but in a more or less unrestricted way.

In this document, we explain two of our current code generators. While the first one is dedicated to Java, based on an open source code generator, and customized for one particular cloud platform, the second one is an extension of a code generator currently available in the Enterprise Architect tool considering C# code generation. Both generators come with dedicated contracts which act as pre-conditions to check the validity of the input models. Here not only the constraints of the UML metamodel are considered, but also the target constraints of the output languages such as naming rules and reserved keywords are checked before the code generation is actually started. By this additional step, many errors such as compile time errors in the generated code can be prevented. Also, due to the need to have some knowledge about the quality of the produced code, we also report on the validation of code generators by reasoning about completeness and correctness of the generators as well as the non-functional properties of the generated code such as readability and maintainability. For this validation concerns, we present a dedicated methodology that could be also used in future generator developments.

1.2 Fitting into overall ARTIST solution

Code generators are used in the second phase of the forward-engineering phase, WP9. As input, they require the models produced in the first phase. Since such models highly depend on the quality of the models produced by WP8, namely the models representing the original application obtained by reserve engineering, then the generation of code also depends on the quality of such models. Consequently, the produced code will preserve all the features and characteristics from the original application only if this information has been preserved in every model obtained during the whole process. For this reason, if there is information that is lost during the process, the user is then responsible for adding this information in the generated code. In fact, within the ARTIST project we also present an approach for moving some information directly from the original code application to the migrated code application, namely the Reusability Trace Tool, also explained in this deliverable. Such tool allows the user to set the criteria for selecting the code he/she thinks appropriate to reuse. After setting the criteria to be applied, an algorithm uses metrics established in the scope of WP5 in order to automatically migrate some code snippets. Furthermore, the user can decide whether to accept the automatic proposal or to make some manual changes.

The results obtained by the prototypes in this deliverable are also closely related to WP11. Indeed, the tools developed in WP11 will take the generated code as input and study different characteristic of it. For instance, it can be studied if the functional behaviour has been preserved. Also, the objectives established in terms of non-functional properties in the context of WP5 must have been achieved, and it is WP11 who deals with this check. Finally, the transformation chain developed in this task, together with the one proposed in [1], has to be well situated in the overall process developed in WP6.

1.3 Innovations

The main innovations in this deliverable are twofold. First, we have contributed first versions of base code generators for Java and C#, the two leading programming languages that we focus on in the ARTIST project. In this regard, we demonstrate how these code generators can be extended and configured for particular cloud platforms. In particular, for obtaining Java code, there are transformation languages, such as Acceleo, that support the implementation of code generators from scratch. In the context of this deliverable, we have extended and

applied the Acceleo-based code generator provided by Obeo Networks, since it provides an extension mechanism that allows specific templates to be plugged in a generic code generator. As for C#, we have dealt with the Enterprise Architect's code generator framework, for which we have shown the mappings between UML and C#. This code generator provides a set of templates that encode a mapping between the UML concepts and the programming languages concepts. As extension points, the generator allows the edition of the templates, so that we obtain the desired mapping.

The second main innovation is a report on the validation of code generators by reasoning about completeness and correctness of the generators as well as the non-functional properties of the generated code such as readability and maintainability. For this, we have employed a dedicated testing framework that is derived from the UML metamodel in order to guarantee that no concept is missing as well as from formal semantic descriptions for UML. Also, we have utilized a formal metric framework that can be used for the generated code.

1.4 Document structure

After this introduction, Section 2 explains some code generation considerations, namely about how to realize different aspects of the code generators. Then, Section 3 presents our approaches for both the UML2Java and UML2C# code generators. After presenting our proposal to validate the code generated by considering functional and non-functional properties in Section 4, Section 5 describes the delivery and usage of our prototypes and Section 6 presents the conclusions of this work.

2 Code Generation Considerations

For realizing code generators by Model-to-Code Transformations, or more generally, by Model-to-Text Transformations (we consider in the context of this document, that code is represented in a literal form), several considerations are needed how to realize different aspects of the code generators. In particular, we aim for a toolbox of code generators which provides generic code generators such as UML2Java and UML2C# which can be the basis for specific extensions that add support for different frameworks and cloud providers.

2.1 Toolbox Global Architecture

In this deliverable we report on our current advances of developing cloud-specific code generators. In particular, we focus on two general purpose programming languages, Java and C#, due to the given use cases in the ARTIST project. First, we aim for reusable and extensible UML2Java and UML2C# translations, before we aim to provide cloud-specific code generators. Thus, we eliminate repetitive implementation efforts for developing specific code generators tailored to specific Java and C# libraries, frameworks, and platforms.

In this respect, we also evaluated the possibility to target higher-level APIs which may allow to generate code which is no longer specific for one cloud provider. In different cloud providers, there can be found similarities among their services. In most cases, especially when dealing with offerings at the level of Infrastructure as a Service, the provided functionalities have many common features, but are accessed and managed via different APIs. In order to bypass this problem and construct a library which will abstract away these differences, there have been some open-source solutions called abstraction libraries. These libraries provide a general view of some of the most commonly provided and used services (mostly the compute, and the object storage service), support different languages and are extensible. This means that anyone can use the provider-specific APIs also provided/supported by these libraries, and create an abstraction layer over a non-frequently used service. These libraries have been investigated in [13], and will be briefly described.

Apache Libcloud[2] is a standard Python library which abstracts away differences among providers and offers a unified API which supports management of resources grouped into four categories: Cloud Servers and Block Storage, Cloud Object Storage and CDN, Load Balancers as a Service, DNS as a Service.

Apache Jclouds[3] is a Java library which provides access to cloud services either as portable abstractions or as cloud-specific features. Jclouds has three levels of abstraction which are summarized by three key-concepts:

- Views, are portable abstractions that are designed to allow developers to write code which uses cloud services while at the same time, being independent from specific vendors. This means that the part of code which accesses a feature of a cloud service is generic and can be applied in cases of different cloud vendors. Apparently the abstraction is only possible in services that share common features among different APIs and cloud providers.
- APIs, in Jclouds represent the actual calls that are made against a specific cloud in order to perform actions. Some APIs are popular and thus are supported by multiple cloud vendors. As an example, EC2 compute API is supported both by Amazon and

[2] https://libcloud.apache.org
[3] http://jclouds.apache.org/

Openstack. So APIs are cloud dependent but not necessarily specific vendor dependent. APIs provide a lower level of abstraction compared to Views.

- Providers, in Jclouds represent instances of a vendor cloud that supports one or more APIs. For example, Amazon vendor offers EC2 through the aws-ec2 provider. Often, the provider is simply the corresponding API plus some instance-specific instantiation values, such as endpoint URLs.

Comparing the three concepts, while going from Views towards Providers, the portability is reduced, while the code expressiveness is increased.

DeltaCloud[4] provides an API server and drivers necessary for connecting to cloud providers forming an abstraction layer over differences among corresponding services. Supported functionalities include:

- Compute service (Instance management/credential)
- Storage Service (Storage volumes/Blob storage)

Deltacloud source code is written in Ruby and communication between Deltacloud server and client is achieved via the Deltacloud REST API or CIMI (Cloud Infrastructure Management Interface) REST API. Any HTTP client can be used but deltacloud provides a Ruby client (the Deltacloud Ruby client) and a C/C++ library (Libdeltacloud Client).

These high-level APIs are mostly focussing on the infrastructure-as-a-service (IaaS)level which we deal in more detail within the context of T9.5 – the deployment of cloud applications to IaaS providers. As long as there are no well-established abstraction APIs for platform-as-a-service (PaaS) providers available, we have to produce cloud provider specific code which is explained in more detail in Section 3 for the Google App Engine.

2.2 Code Generation Frameworks – State of the Art

Before we explain how we developed our current code generators, we survey different approaches for developing model-to-text transformations in order to select the most appropriate approach for our purposes. MDE uses models as first-class-entities for software development. Models are conceptual representations of software applications or their parts, for different aspects, and with different levels of abstraction, ranging from computation independent (CIM) to platform independent (PIM) to platform specific (PSM). At some point in the MDE-based software engineering life cycle, PSMs require to be transformed into linearized text representations of software artefacts, such as code, deployment specifications, documentation, etc. The transition between PSMs and their corresponding software textual serializations can be automated by applying Model-to-Text (M2T) transformations.

Different formal specifications and supporting tools for M2T transformations have been proposed and implemented. In the remainder of this section, we survey those M2T transformation approaches which are compatible with the ARTIST tooling baseline, based on Eclipse Modeling Framework and Enterprise Architect.

MOFM2T[5]: MOFM2T is a M2T language specification created by OMG[6] that enables the definition of M2T transformations between MOF[7] models and textual representations. The

[4] https://deltacloud.apache.org/
[5] http://www.omg.org/spec/MOFM2T/1.0/
[6] www.omg.org

MOFM2T language specification is template-based. A template defines a target text template that includes placeholders that are parameterized to be replaced with matching model elements. Placeholders are defined using query expressions on elements of the meta-model. Matching model elements are expanded in the text template to produce the target text.

MOFM2T facilitates template compositions (i.e. template invocation) and module organization to manage complex M2T transformations. Different implementations of the MOFM2T specification have been produced by different vendors (see below some of them).

Xpand/Xtend [8]: Xpand is a template-based (i.e., similar to JSP) M2T transformation framework for EMF models, included in the Eclipse M2T project. It offers advanced template based code generation features such as type safety and polymorphic dispatch (appropriate template selection for model elements types at runtime). The Xpand language has evolved into the more general Xtend language which now subsumes the functionalities of Xpand and compiles directly to Java which allows for an easier integration with Java-based frameworks.

Acceleo[9]: Acceleo is an open source implementation of MOFM2T, an EMF[10] M2T code generator framework, included in the Eclipse Modeling release. As an implementation of MOFM2T, Acceleo M2T language is template based. Acceleo uses OCL[11] as model querying language. Hence, Acceleo M2T language can be considered a superset of OCL, facilitating its learning to modelers familiarized with MDE standards. Templates support both dynamic and static code and the specification of user code blocks (e.g. protected areas) to avoid overriding code added manually. Acceleo offers a complete M2T framework that includes the Acceleo M2T language, a runtime and SDK development tools. This SDK is included as part of the Eclipse M2T project. These SDK tools include advanced editors for Acceleo modules (including code completion, validation, quick fixing, pattern management, etc.), wizards to create modules from existing code (bottom-up approach) and support for debugging, profiling and launching Acceleo modules on selected input models. There is a repository of Acceleo modules[12], including the UML 2.1 to Java module.

MOFScript[13]: MOFScript is one of the first M2T framework implementation of the OMG MOFM2T specification and it is compatible with the OMG M2M transformation specification: QVT[14]. It is characterized by its simplicity, since the language has few constructs and it is similar to existing imperatives scripting programming languages. Besides, language rules are explicitly executed sequentially. It supports traceability between models and generated text. The MOFScript SDK is included in the Eclipse M2T project. This SDK includes the language metamodel, a parser, a runtime environment and a module editor that offers advanced editing features such as code assist. MOFScript is compatible with input EMF model instances of MOF, UML, Ecore or other EMF-based metamodels.

JET[15]: JET another framework for M2T transformations included in the Eclipse M2T project. However, this project has not released new versions of this framework since 2011. Its

[7] MOF [www.omg.org/mof/] is an OMG specification of standard interfaces that enables the definition and manipulation of a set of interoperable meta-models and their corresponding model instances.
[8] http://wiki.eclipse.org/Xpand, http://www.eclipse.org/xtend/
[9] http://www.eclipse.org/acceleo/
[10] Eclipse Modeling Framework
[11] Object Constraint Language
[12] www.acceleo.org/pages/modules-repository/en
[13] http://www.eclipse.org/gmt/mofscript/
[14] www.omg.org/spec/QVT/1.1/
[15] http://eclipse.org/modeling/m2t/?project=jet#jet

approach is based on JSP-like templates that can be edited and transformed into textual artefacts such as source code (i.e. Java, HTML, etc.) or configuration files (i.e. properties, XML, etc.)

Code Template Framework (CTF): Enterprise Architect from Sparx Systems provides a Code Template Framework (CTF) which can be used for:

- Forward Engineering of UML models
- Behavioral Code generation
- Model Driven Architecture (MDA) Transformations

When forward engineering a Class model, the code templates define how the code is to be generated for a given programming language. The templates are written as plain text with a syntax that shares some aspects of both mark-up languages and scripting languages. Enterprise Architect provides a set of Base Templates that you can add to or adapt if required.

In addition, Enterprise Architect supports user-definable code generation of the UML behavioral models. This uses the standard CTF but includes specific Enterprise Architect Simulation Library (EASL) code generation macros.

The base templates form a hierarchy, which varies slightly across different programming languages. In a typical template hierarchy relevant to a language like C# or Java (which do not have header files) the templates can be modeled as Classes, but usually are just plain text. This hierarchy would be slightly more complicated for languages like C++ and Delphi, which have separate implementation templates. Each of the base templates must be specialized to be of use in code engineering; in particular, each template is specialized for the supported languages (or “products”).

For example, there is a *ClassBody* template defined for C++, another for C#, another for Java, and so on. By specializing the templates, you can tailor the code generated for the corresponding UML entity. Once the base templates are specialized for a given language, they can be further specialized based on:

- A Class's stereotype
- A feature's stereotype (where the feature can be an operation or attribute)

This type of specialization enables, for example, a C# operation that is stereotyped as «property» to have a different Operation Body template from an ordinary operation; the Operation Body template can then be specialized further, based on the Class stereotype.

CodeSmith: CodeSmith Generator is a template-driven source code generator for different ASCII-based (programming) languages such as C#, Java or VB. The syntax of the templates is based on ASP.NET. Basically, XSD schemas are used for transforming its instances as XML documents to source code.

T4 or Text Template Transformation Toolkit: T4 text templates, which can be created using Visual C#, can be used to generate any text file or language. A template may consist of a mixture of text blocks and control logic. As input an XML document can be used. In addition, also diagrams of workflow in a business activity can be used for generating code. However, if your code generator is based on T4 templates, Visual Studio has to be installed, not only for defining the templates but also for executing the code generation based on the templates.

My 2nd Generation Code Generator: My2ndGeneration is a free .Net development tool for generating source code based on templates. These templates can be defined in JScript,

VBScript, C# or VB.NET. Databases such as SQL Server, PostgreSQL or Oracle are used as input for code generation.

Synopsis. As can be seen by comparing the descriptions of the different code generation implementation languages, they are mostly all based on templates which query some kind of information resource, in our setting the models, to produce the required code. Based on the experiences of the ARTIST members and tool integration capabilities, we base our code generation development on Acceleo inside Eclipse and CTF inside Enterprise Architect.

2.3 Traceability from Model to Code and Back Again – Reuse of Existing Code

Another important aspect we like to explicitly discuss is the traceability between model and code in the context of code generation. In particular, for debugging purposes, this kind of tool support is of paramount importance as well as for change management when the code or the models have to evolve. An Eclipse-based tool has been developed allowing reusing the code of the system to be migrated in a non-intrusive way, i.e., without modifying the models generated all along the process. The Reusability trace tool (RTT) allows the user to set the criteria for selecting the code he/she thinks appropriate to reuse. After setting the criteria to be applied, an algorithm is executed. This algorithm uses metrics established in the scope of WP5 and, when finished, the user sees on a window the set of code snippets automatically proposed to be reused (class and method names). Through this result window, the user is able to decide whether to accept the automatic proposal or to make some changes by selecting/deselecting the corresponding method names (via check boxes). The finally selected method names are registered in a configuration file.

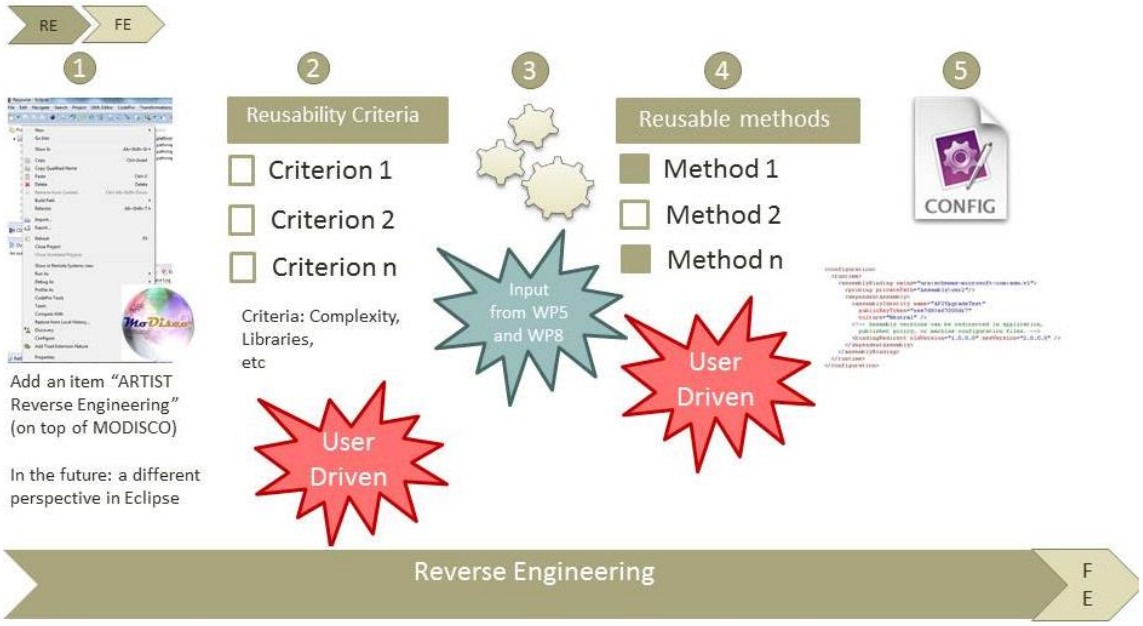

Figure 2 Traceability in Reverse Engineering with RTT

Inside forward engineering tools (WP9) integrated in Eclipse, it is implemented a mechanism that allows the user to rewrite the code generated with M2M and M2T transformations by accessing directly the source code and the list of reusable methods in the previously produced configuration file.

The idea is to inject the original code directly on those parts of the application that cannot be generated through forward engineering techniques.

The resulting code (forward generated + injected code) is then commented and marked with "Todo" tag depending on whether that piece of code has been selected as reusable or not by the RTT (according to the migration's technological nature).

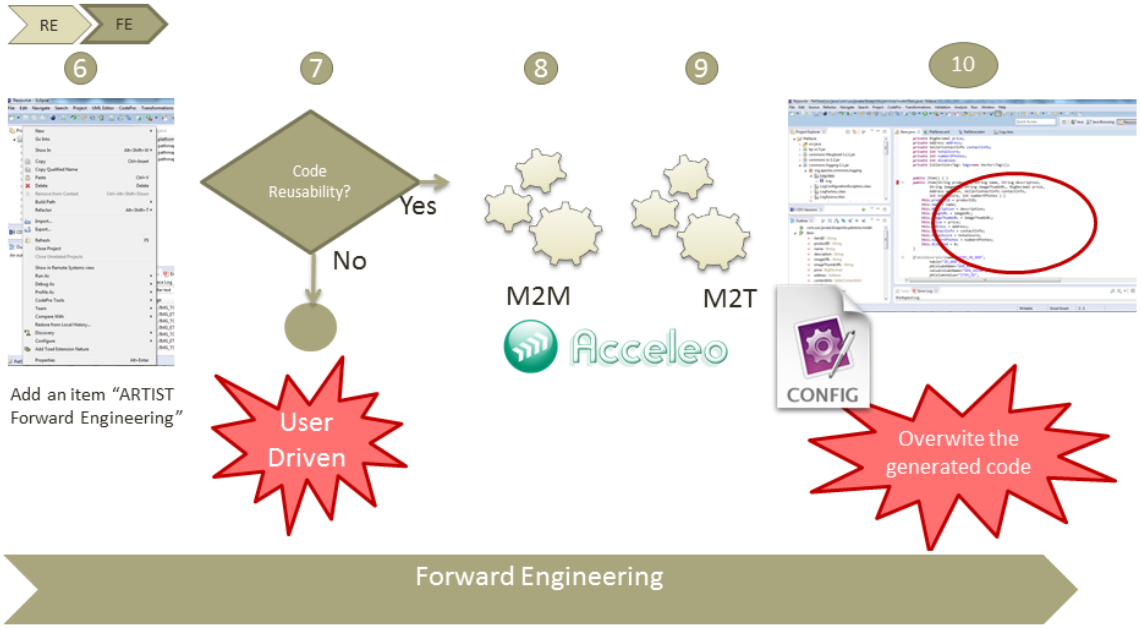

Figure 3 Traceability in Forward Engineering with RTT

Technical details and a user manual can be found in section 3.4 of [14].

2.4 User Interaction: Code Generator Configurations and Extensions

There are several approaches for configuring code generators in a black-box manner:

1) Application of dedicated (UML) profiles to the model which drive the code generation. For instance, consider the application of stereotypes marking some UML classes as entity or service classes. Then, code generators can provide specialized functions to react in the code generation to produce Entity Beans or Service Beans in cases for J2EE code is generated.
2) Configuring some global settings for the code generator. For instance, before generating code with Enterprise Architect, several settings can be specified. Some of the global settings are depicted in Figure 4.

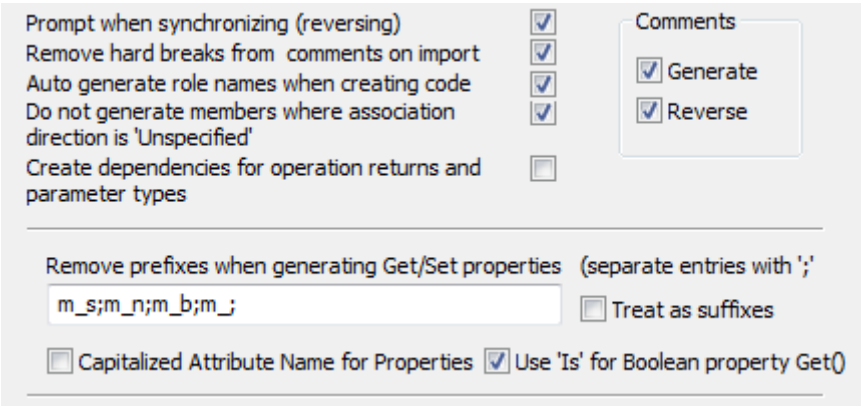
Prompt when synchronizing (reversing)
Remove hard breaks from comments on import
Auto generate role names when creating code
Do not generate members where association direction is 'Unspecified'
Create dependencies for operation returns and parameter types
Comments
Generate
Reverse
Remove prefixes when generating Get/Set properties (separate entries with ';'
m_s;m_n;m_b;m_;
Treat as suffixes
Capitalized Attribute Name for Properties
Use 'Is' for Boolean property Get()

Figure 4 Configuration of global settings for the code generator

For example, the user can specify whether or not comments or role names are generated when using associations. Furthermore, the user can specify whether EA should automatically create “constructors” or “destructors” when generating code (cf. Figure 5).

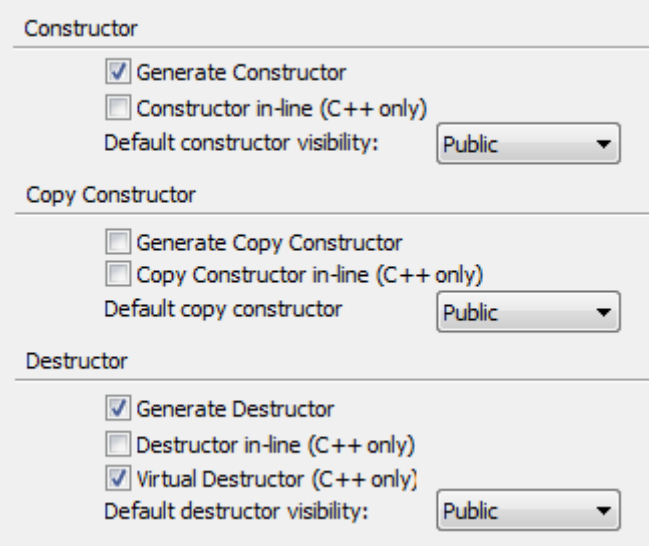

Figure 5 Options for the code generator

Other programming language specific parameters can be set by the user (cf. Figure 6). For example, custom collection classes can be defined, when multiplicities for associations are used in the UML models. Custom collection classes can be simple substitutions (such as *CArray<#TYPE#>*) or a mix of other strings and substitutions (such as *Cmap<CString,LPCTSTR,#TYPE#*,#TYPE#*>*). The following collection classes are defined by default: *List<#TYPE#>;Stack<#TYPE#>;Queue<#TYPE#>.*

There are also white-box approaches to configure and extend code generators such as template inheritance and template aspects, which may complement the discussed black-box approaches. In the course of the ARTIST project, we explore how to best combine the different approaches in order to realize highly flexible code generators: (i) dedicated requirements based on the developers preferences, how the code should look like, e.g. preferred collections types should be configurable in the base code generations for Java and C# using the black-box configuration approaches, and (ii), these code generators should be adaptable to specific platforms such as cloud providers by applying more powerful but at the same time more challenging white-box extension concepts such as template inheritance to name the most frequently used extension kind.

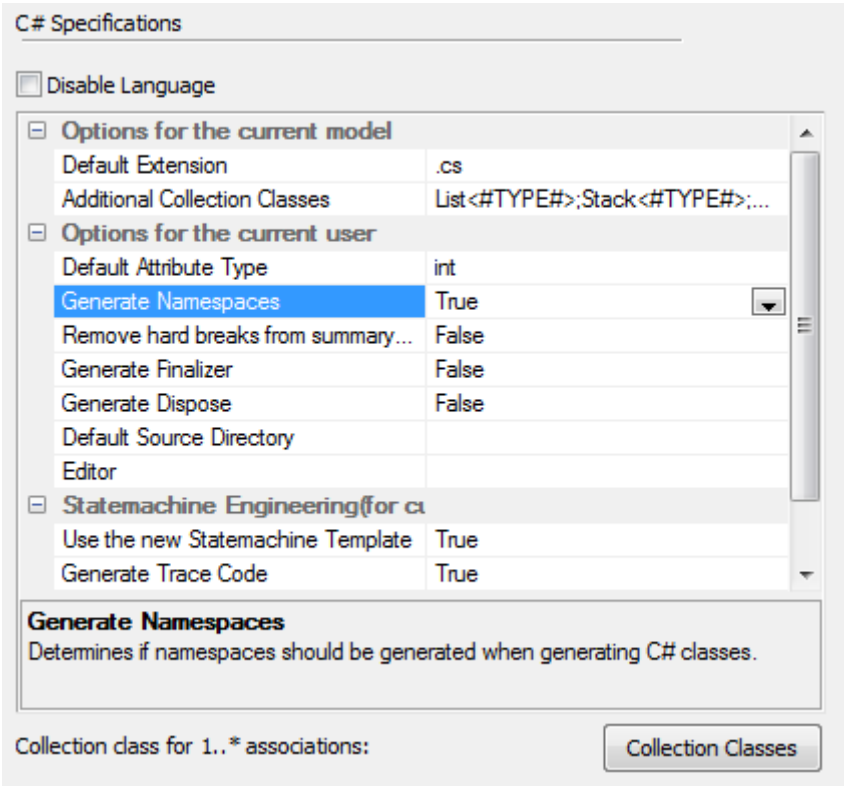

Figure 6 Programming language specific parameters for the code generator

3 Implementation

3.1 Functional description

This section describes the code generators we are currently implementing. Because the ARTIST use cases are either Java-based or C#-based, we develop two code generators to reach these platforms from our modeling approaches. Both code generators consider the UML class diagram language of the UML language family and follow the same design rationales:

1) **Contract-based code generators**: Because UML is a large language and specific target platforms impose specific constraints on the models, we follow a contract-based definition of the code generators which provides two main advantages. First, the contracts may be used to filter out invalid input to the code generator, i.e., input which would lead to invalid output. Second, we can employ the contracts to test the code generators by establishing a test suite where the contracts are used as test oracles for the test runs.
2) **Open, extensible, and configurable code generators**: Because there may be specific requirements imposed by reusing specific APIs, frameworks or by following different coding styles, etc., the code generators should be open, i.e., they should be modifiable for specific projects and they should provide specific extension mechanisms in order to allow a more systematic customization of the code generators. Finally, the code generators should provide configuration parameters in order to switch between target alternatives in order to ensure the applicability of the base code generators.
3) **Semantic-preserving code generators**: Because the UML class diagram language is not only able to define the shape of UML classes, i.e., features and methods, but at the same time represents a constraint language, e.g., consider the multiplicities, different association types, the code generators should not only generate the blueprint of the classes such as fields and operation skeletons, but should provide consistency preserving operation implementations in order to ensure that no invalid object configurations are constructed on the code level.
4) **Understandable code generation outputs**: Finally, the code generators should produce source code which is still accessible by the programmers. In the best case, the programmers should not be able to distinguish if the code is manually produced or if the code has been generated. Thus, the generated code has to be readable, understandable, etc., in summary all properties should be fulfilled which are expected from manually produced code. This requirement is in particularly important when the code has to be completed by programmers.

3.2 UML2Java Code Generator – Technical Description

Today's UML tools provide code generation facilities to produce program code from UML models, in particular if the targeted programming language is Java. On the other hand, transformation languages, such as Acceleo, support the implementation of code generators from scratch. In the context of this deliverable, we extended and applied the Acceleo-based code generator provided by Obeo Networks[16]. The main reasons behind this selection are twofold. First, Acceleo is a reference implementation of OMG's MOF Model to Text Language (MTL) standard. And, second, it provides an extension mechanism that allows specific templates to be plugged in a generic code generator. In the ARTIST project, the idea is to implement such extensions for specific targets of the code generation. For instance, when

[16] http://marketplace.eclipse.org/content/uml-java-generator/

targeting the datastore of the Google App Engine, specific program code for the Objectify[17] framework to access this datastore can be produced by an extension. Clearly, such extensions are realized on top of a generic code generator that implements the mapping between UML and Java without considering framework-specific program code. In Section 3.2.1, we give an architectural overview of the code generator facility used in the ARTIST project for producing Java program code from UML models. Thereby, we set the focus on the structural perspective of applications modelled in UML. Then, in Section 3.2.2, we discuss in more detail the generic part of the code generator and an extension that is specific to the Objectify framework. Finally, in Section 3.2.3, we give a summary of the technologies used to realize the *UML2Java* code generator and clearly point out our extensions developed for the Acceleo-based code generator developed by Obeo Networks.

3.2.1 Prototype architecture

UML Models are injected to the *Code Generator* as a basis to generate the corresponding *Program Code*. In addition, a *Generator Configuration* enables parameters to be passed to the code generator. A configuration is usually required to define Java-specific implementation details, such as the collection types that are expected to be used in the extracted program code, and project-specific details that refer to the employed development environment, such as the Eclipse environment. For instance, the Eclipse environment provides several different project types to realize a Java application. Clearly, in the development of such applications, frameworks play a central role. Hence, code generators need to be extensible and flexible enough to extract program code that uses a selection Java *Framework.* In this respect, *UML Profiles* are applied to support such Java frameworks also at the modeling level. A detailed discussion of how such UML profiles are generated is given in [15] while their application in the context of forward engineering is discussed in [1]. *Code Generator Extensions* rely on UML profiles as they get triggered if certain stereotypes have been applied to model elements. In this sense, such extensions provide code templates that correspond to stereotypes of UML profiles, which in turn refer to a Java framework. As a result, extensions pave the way for code generators to extract richer program code from profiled UML models, i.e., a model to which a UML profile has been applied. For instance, complete method bodies for CRUD operations can be generated for domain classes as they can be indicated by the respective stereotypes. Figure 7 summarizes the architectural overview of the *UML2Java* code generation facility that is developed for the purpose of extracting Java program code from injected UML class models.

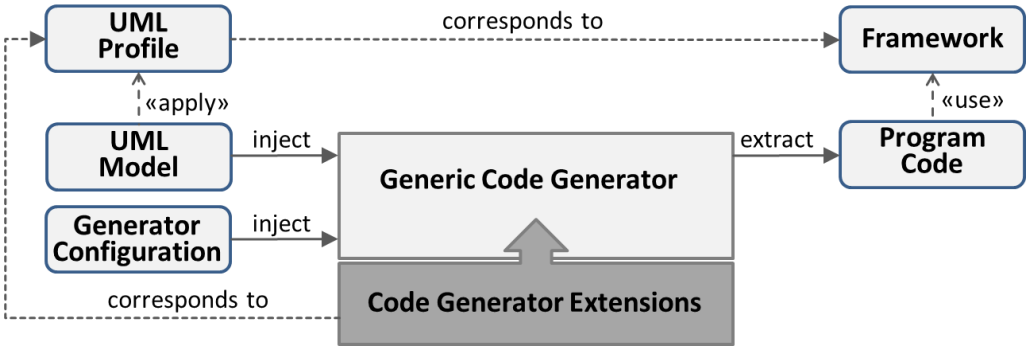

Figure 7 Architectural Overview of the UML2Java Code Generation Facility

[17] https://code.google.com/p/objectify-appengine/

3.2.2 Components description

The *UML2Java* code generator is realized on the basis of three main components. The *Generic Code Generator* component implements a mapping from UML to Java for the main concepts of the UML class diagram language whereas the *Code Generator Configuration* component is dedicated to support the passing of user parameters to the code generator. Finally, the *Code Generator Extensions* component is considered as a collection of extensions provided for the generic code generator component.

3.2.2.1 Generic Code Generator

The generic code generator realizes a basic mapping between UML concepts and Java concepts. Table 1 gives an overview of the currently provided concepts of the two languages. The idea is to significantly improve the current mapping and its realization based on our lessons learned from the evaluation of the current code generation support. Results gained in this respect are reported in [16].

Table 1 Mapping between UML Concepts and Java Concepts

UML Concept	Java Concept
Association	If the member ends of the association are contained by the related classes, the association is translated into Java fields and optional accessor methods
Class	Java class. Visibility and Modifiers are supported.
Comment	Comments for Java elements.
Element Import / Package Import	Required import statements for Java classes.
Enumeration	Java enum type. Visibility is supported.
Enumeration Literal	Java fields of enum type.
Generalization	Extensions between Java classes / interfaces.
Interface	Java interface. Visibility is supported.
Interface Realization	Java classes that implement Java interfaces.
Operation	Java method. Visibility, Modifiers, and Return type are supported.
Package	Java package. Visibility is supported.
Parameter	Java parameter. Modifiers are supported.
Property	Java field. Visibility, Modifiers and Types are supported.
Stereotype	Applied Stereotypes are translated into Java annotations.

3.2.2.2 Code Generator Configuration

The main purpose of the configuration component is to allow the user to adapt the imposed conventions of the generic code generator.

- Project-specific information, such as Java execution environment, source folder and output folder, and Eclipse project type
- Field accessor support, such as getter / setter methods and advanced accessors for collections that include methods to add elements to and remove elements from collections
- Collections types to realize multi-valued fields in Java programs

3.2.2.3 Code Generator Extensions

With extensions to the generic code generator, framework-specific program code can be produced. We have realized an extension for the Objectify framework. Basically, this extension provides templates that produce an Objectify-specific persistency layer.

- Objectify-specific accessor methods that deal with key-based references
- Service class that provide CRUD-based access to Objectify entities
- Bodies of CRUD-based methods
- Registration of Objectify entities

Stereotypes of the UML profile for Objectify are used to annotate the entities and identifiers. The idea is to realize additional extensions that deal with UML profiles presented in D9.3. Again, the gained results in this respect are presented in [16].

3.2.3 Technical specifications

The *UML2Java* code generator uses the following components and technologies. It is important of note that we extended the Acceleo-based code generator developed by Obeo Networks for the needs of the ARTIST use cases. First, we have added generic support for UML profiles. Thereby, UML profiles are considered as annotation mechanism. In this sense applied stereotypes at the modeling level are considered as Java annotations at the programming level. Second, we demonstrated how the generic code generator can be extended. In this respect, we have implemented an extension for the Objectify framework.

- **Generic Code Generator**: We have added support for UML profiles to the Acceleo-based code generator developed by Obeo Networks. In this respect, we have also restructured the code templates as we produce Java annotations for corresponding UML stereotypes.
- **Code Generator Extension for Objectify**: We have implemented an Acceleo-based code generator extension for the Objectify framework.
- **Code Generator UI**: Code generator UI developed by Obeo Networks.
- **Acceleo**: Base technology to realize the code generation facility
- **Java**: Base technology to realize the code generation facility
- **UML**: Modeling language to represent the models and profiles injected to the code generation facility

3.2.4 User Manual

To demonstrate how the *UML2Java* code generator can be used, we use the Java Petstore reference application as example. In fact, we consider the objectified backend of the Java Petstore as described in [1] (see Figure 8 and Figure 9). Hence, the input for the code

generator refers to the model that represents the domain classes and the respective service classes to access them. It is platform-specific in the sense that it refers to the Objectify framework. As a result, not only the generic code generator is executed but also the extension regarding Objectify. Because extensions are implemented in terms of Eclipse plugins, their execution is triggered by the Acceleo engine.

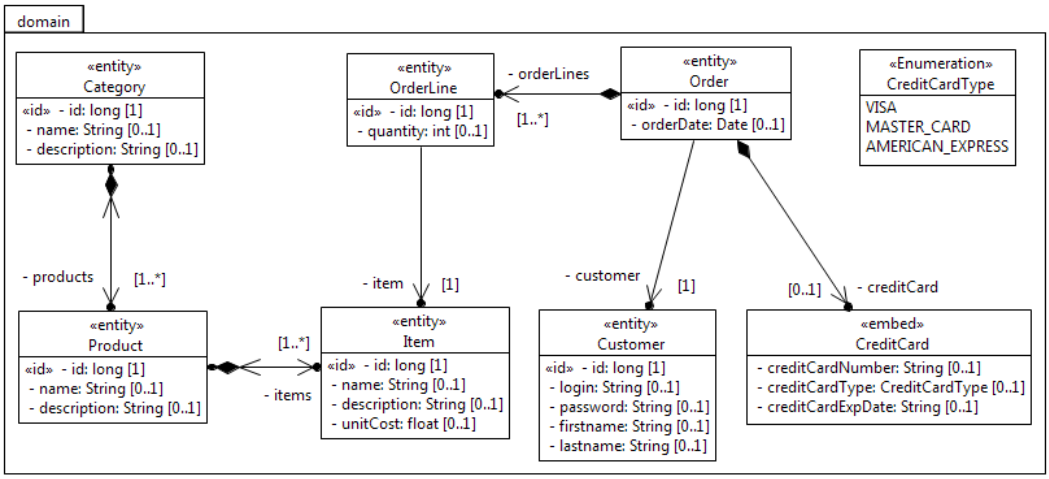

Figure 8 Objectified Domain Model (PSM)

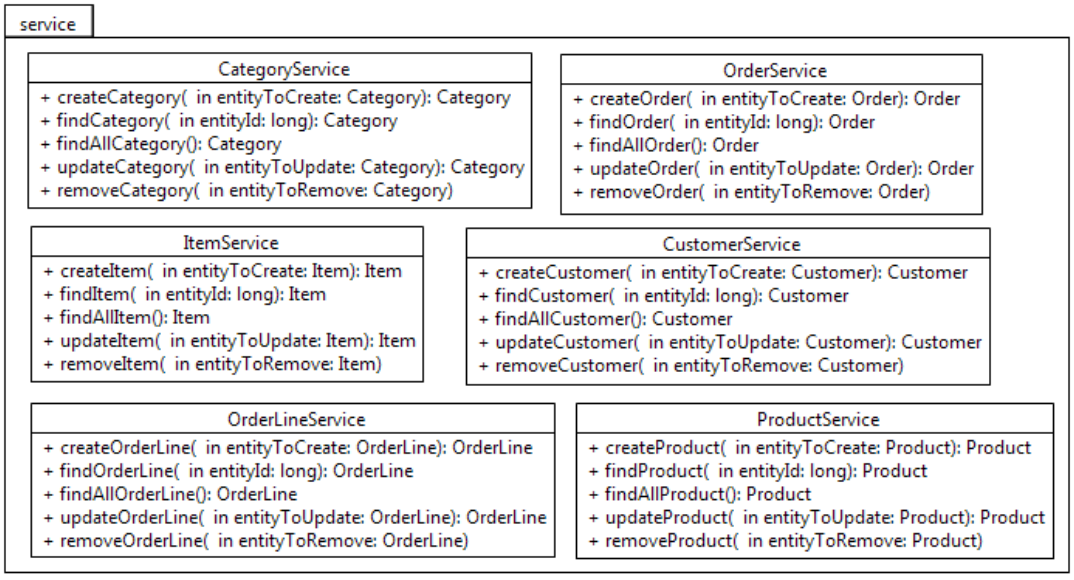
service
CategoryService
+ createCategory(in entityToCreate: Category): Category
+ findCategory(in entityId: long): Category
+ findAllCategory(): Category
+ updateCategory(in entityToUpdate: Category): Category
+ removeCategory(in entityToRemove: Category)
OrderService
+ createOrder(in entityToCreate: Order): Order
+ findOrder(in entityId: long): Order
+ findAllOrder(): Order
+ updateOrder(in entityToUpdate: Order): Order
+ removeOrder(in entityToRemove: Order)
ItemService
+ createItem(in entityToCreate: Item): Item
+ findItem(in entityId: long): Item
+ findAllItem(): Item
+ updateItem(in entityToUpdate: Item): Item
+ removeItem(in entityToRemove: Item)
CustomerService
+ createCustomer(in entityToCreate: Customer): Customer
+ findCustomer(in entityId: long): Customer
+ findAllCustomer(): Customer
+ updateCustomer(in entityToUpdate: Customer): Customer
+ removeCustomer(in entityToRemove: Customer)
OrderLineService
+ createOrderLine(in entityToCreate: OrderLine): OrderLine
+ findOrderLine(in entityId: long): OrderLine
+ findAllOrderLine(): OrderLine
+ updateOrderLine(in entityToUpdate: OrderLine): OrderLine
+ removeOrderLine(in entityToRemove: OrderLine)
ProductService
+ createProduct(in entityToCreate: Product): Product
+ findProduct(in entityId: long): Product
+ findAllProduct(): Product
+ updateProduct(in entityToUpdate: Product): Product
+ removeProduct(in entityToRemove: Product)

Figure 9 Generated Service Classes (PSM)

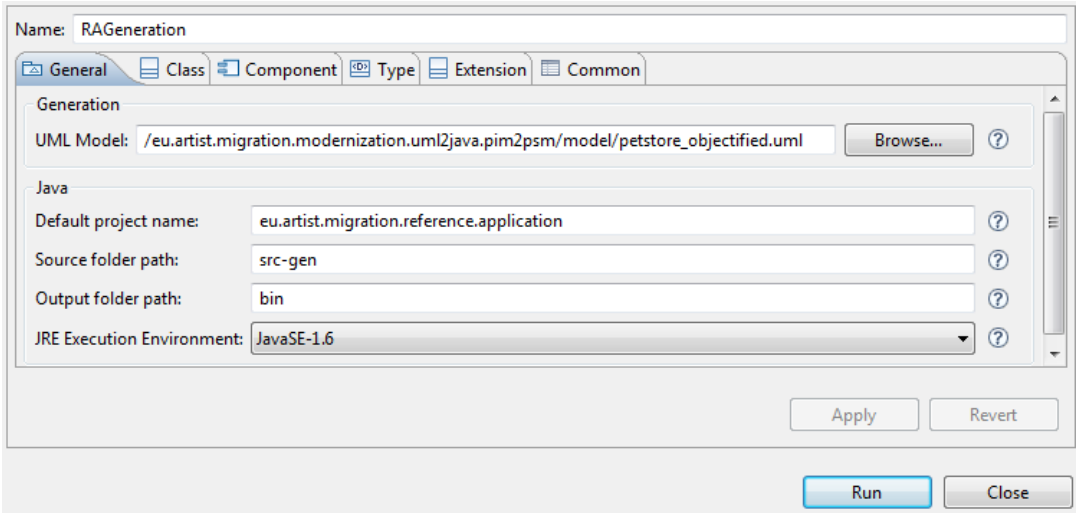

Figure 10 Run Configuration of *UML2Java* Code Generator

Hence, there is no user interaction required to execute extensions to the *UML2Java* code generator. Figure 10 depicts the "run configuration" for executing the *UML2Java* code generator. It allows the user to select the input model for the code generation and the target project to which the produced program is extracted. The listing beneath shows an excerpt of the produced program code for the `Order` class.

```
package org.agoncal.application.petstore.domain;

import java.util.ArrayList;
import com.googlecode.objectify.Ref;
import com.googlecode.objectify.annotation.Id;
import com.googlecode.objectify.annotation.Entity;
import java.util.List;
import org.agoncal.application.petstore.domain.OrderLine;

@Entity
public class Order {
	public List<Ref<OrderLine>> orderLines;

	/**
	 * Returns orderLines.
	 * @return orderLines
	 */
	public List<Ref<OrderLine>> getOrderLines() {
		return this.orderLines;
	}

	/**
	 * Adds a value to attribute orderLines.
	 * @param addOrderLines
	 */
	public void addOrderLines(OrderLine entity) {
		if (orderLines == null)
			orderLines = new ArrayList<Ref<OrderLine>>();
		orderLines.add(Ref.create(entity));
	}
```

An excerpt of the respective service class is shown in the next listing. The service class basically provides CRUD operations for the `Order` entity.

```
package org.agoncal.application.petstore.service;

import static com.googlecode.objectify.ObjectifyService.ofy;
import java.util.List;
import org.agoncal.application.petstore.domain.Order;

public class OrderService {
	/**
	 * Description of the method createOrder.
	 * @param entityToCreate
	 */
	public Order createOrder(Order entityToCreate) {
		if (entityToCreate != null) {
			// TODO: Provide business logic if required
			// finally the entity is stored - synchronous approach
			ofy().save().entity(entityToCreate).now();
		} else {

			// TODO: Exception handling if the entity is null
		}

		return entityToCreate;
	}

	/**
	 * Description of the method findAllOrder.
	 */
	public List<Order> findAllOrder() {
		return ofy().load().type(Order.class).list();
	}
```

3.3 UML2C# Code Generator – Technical Description

Enterprise Architect's code generation framework (CGF) is based on the code template framework (CTF). The basics of the CTF are described in previous sections. The CTF provides a set of templates for each supported programming language. A mapping between the UML concepts and the programming language concepts are encoded in the code templates.

General code conventions for C#

- Namespaces are generated for each package below a namespace root
- The *Const* property of an attribute corresponds to the read-only keyword, while the tag *const* corresponds to the *const* keyword
- The value of *inout* for the *Kind* property of a parameter corresponds to the ref keyword
- The value of out for the *Kind* property of a parameter corresponds to the *out* keyword
- Partial Classes can be modelled as two separate Classes with the *partial* tag
- The *IsLeaf* property of a Class corresponds to the *sealed* keyword

Stereotype Conventions for C#

Stereotype	Applies To	Corresponds To
enumeration	Class	An *enum* type.

event	Operation	An event.
extension	Operation	A Class extension method, represented in code by a *this* parameter in the signature.
indexer	Operation	A property acting as an index for this Class.
partial	Operation	The *partial* keyword on an operation.
property	Operation	A property possibly containing both read and write code.
struct	Class	A *struct* type.

Tagged Value Conventions for C#

Tag	Applies To	Corresponds To
argumentName	Operation with stereotype *extension*	The name given to *this* parameter.
attribute_name	Operation with stereotype *property* or *event*	The name of the variable behind this property or event.
className	Operation with stereotype *extension*	The Class that this method is being added to.
const	Attribute	The *const* keyword.
definition	Operation with stereotype *partial*	Whether this is the declaration of the method, or the definition.
delegate	Operation	The *delegate* keyword.
enumType	Operation with stereotype *property*	The datatype that the property is represented as.
extern	Operation	The *extern* keyword.
fixed	Attribute	The *fixed* keyword.
generic	Operation	The generic parameters for this Operation.
genericConstraints	Templated Class or Interface, Operation with tag *generic*	The constraints on the generic parameters of this type or operation.
Implements	Operation	The name of the method this implements, including the interface name.
ImplementsExplicit	Operation	The presence of the source interface name in this method declaration.
initializer	Operation	A constructor initialization list.
new	Class, Interface, Operation	The *new* keyword.
override	Operation	The *override* keyword.
params	Parameter	A parameter list using the params keyword.
partial	Class, Interface	The *partial* keyword.
readonly	Operation with stereotype *property*	This property only defining read code.

sealed	Operation	The *sealed* keyword.
static	Class	The *static* keyword.
unsafe	Class, Interface, Operation	The *unsafe* keyword.
virtual	Operation	The *virtual* keyword.
writeonly	Operation with stereotype *property*	This property only defining write code.

3.3.1 Prototype architecture

The current C# code generator for ARTIST is based on Enterprise Architect, EA's code template framework and an additional model validation component.

Model Adaptation for reusing the Code templates. Basically, the CTF is used within ARTIST for generating C# source code. Adaptation of the code generation can be done by changing the templates themselves. In addition, an EA plugin has been implemented to pre-process the model. That means that the model, for which code is generated, can be adapted following the needs of ARTIST. With this approach the standard code templates can be reused in general.

Model Validation. Before code is generated, the model can be validated to find possible flaws at an early stage. Checks on a syntactic level are implemented to generate "compilable" code. This component is also developed as plugin extension for Enterprise Architect. It is based on a model validation framework which allows for adding new rules in a very quick manner. In addition, the plugin provides the ability to auto-correct flaws in the model.

3.3.2 Components description

The C# code generator adapted for ARTIST consists of two components: First, a model validation component and, second, the code generator plug-in using the code template framework.

In order to generate valid C# code, the model is validated before it is generated. During this validation process, violations are resolved automatically, semi automatically or manually (cf. Figure 11).

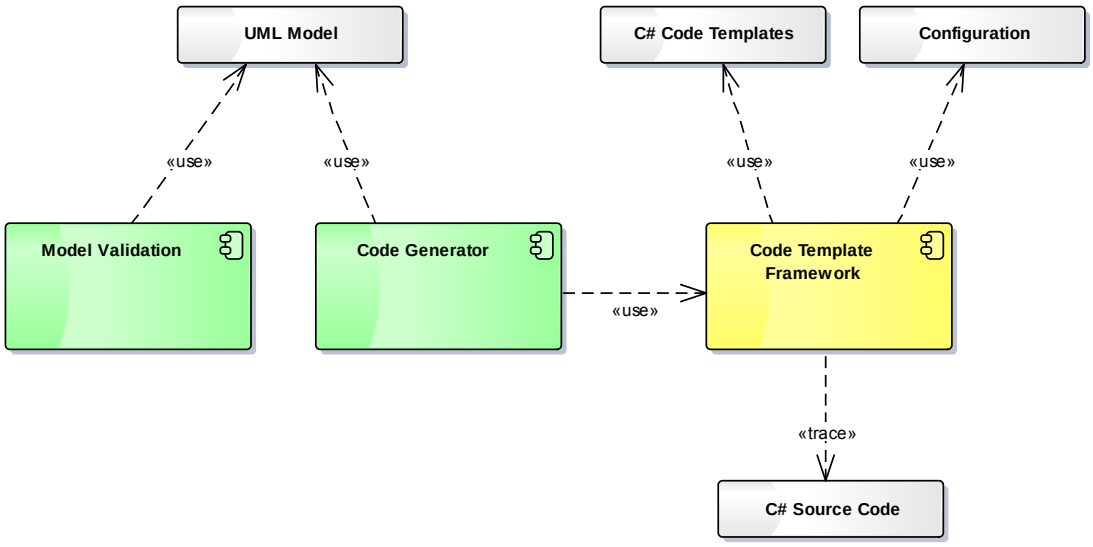

Figure 11 C# Validation process

The diagram in Figure 12 represents the code generation process. To generate code from an UML model, the user has to select a UML package which contains classes and behavioural models like activity models and state machines. When the user triggers the code generation, the model validation component validates the content of the selected package based on C# validation rules. In case a violation is detected, the model validation component provides one or more suggestions to solve the issue. The user can select one suggested correction. Some rules may be configured to be auto solved, so no user interaction is necessary. Alternatively, the user can correct the model manually. After the model is changed, the validation starts again until no more violations are found. The next step is the actual process of writing C# code based on the model and the predefined configuration. In this architecture, the model is validated and corrected before the code is generated. This approach was selected due to the possibility to reverse engineer the generated code. In case the violations are corrected during the code generation process, a reverse engineering of the generated code would change the model and the user may be confused about this changes. With the approach of a pre validation, the user is involved in the correction process.

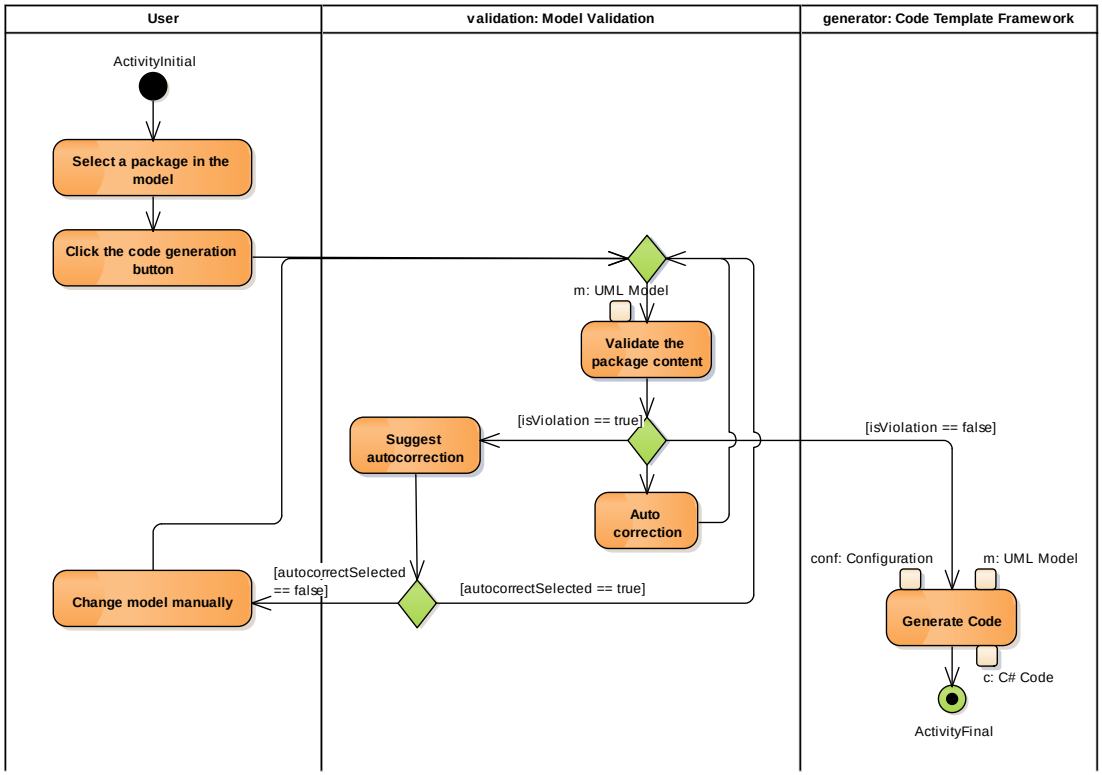

Figure 12 C# Code Generation Process

3.3.3 Technical specifications

The system requirements are the following:

- Windows Operating System
- Microsoft .Net 3.5
- Optional DBMS (MSSQL, Oracle, MySQL, Postgres, Firebird)

3.3.4 User Manual

Before generating code out of an EA model, it can be validated to produce compilable source code. For this purposes, the Artist EA add-in has to be installed (ArtistSetup.msi in GitHub). You can verify the correct installation under “Programs and Features” if you work with Windows (cf. Figure 13).

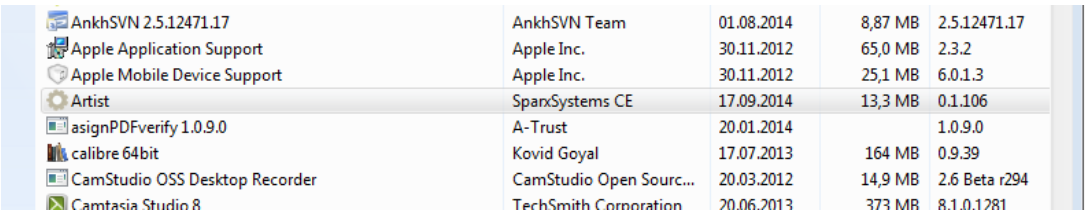
AnkhSVN 2.5.12471.17 AnkhSVN Team 01.08.2014 8,87 MB 2.5.12471.17
Apple Application Support Apple Inc. 30.11.2012 65,0 MB 2.3.2
Apple Mobile Device Support Apple Inc. 30.11.2012 25,1 MB 6.0.1.3
Artist SparxSystems CE 17.09.2014 13,3 MB 0.1.106
asignPDFverify 1.0.9.0 A-Trust 20.01.2014 1.0.9.0
calibre 64bit Kovid Goyal 17.07.2013 164 MB 0.9.39
CamStudio OSS Desktop Recorder CamStudio Open Sourc... 20.03.2012 14,9 MB 2.6 Beta r294
Camtasia Studio 8 TechSmith Corporation 20.06.2013 373 MB 8.1.0.1281

Figure 13 Verifying correct installation of Artist EA add-in

If the add-in is loaded successfully you can find a dedicated menu entry under “Extensions” inside Enterprise Architect. For activating or deactivating validation rules you can click on “Extensions -> Artist -> Options” (Figure 14).

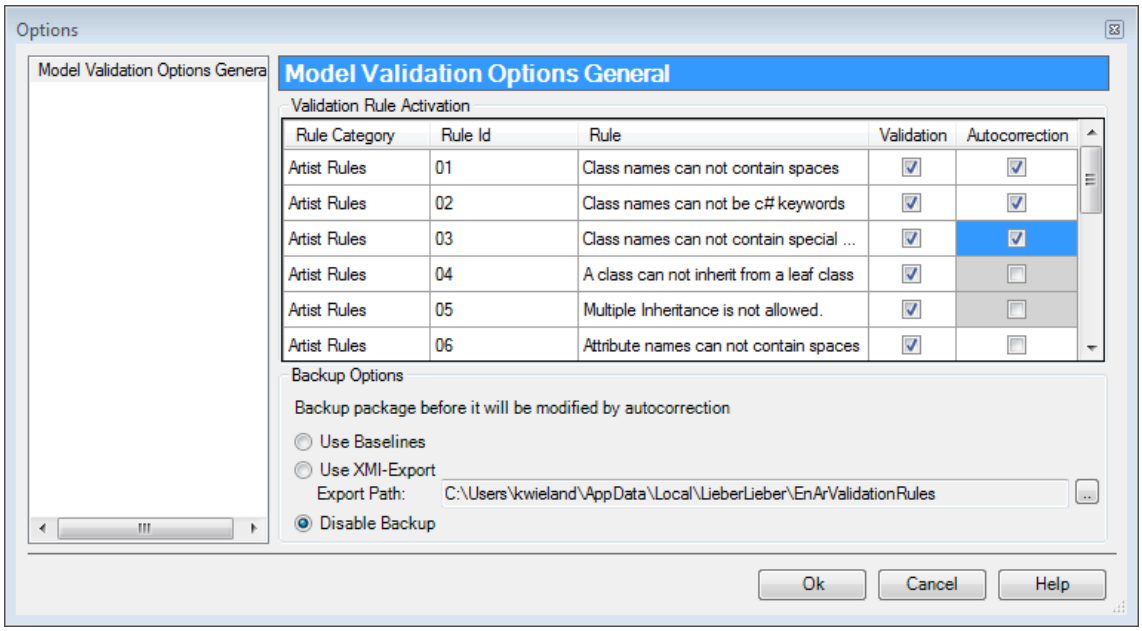
Options
Model Validation Options Genera
Model Validation Options General
Validation Rule Activation
Rule Category
Rule Id
Rule
Validation
Autocorrection
Artist Rules 01 Class names can not contain spaces
Artist Rules 02 Class names can not be c# keywords
Artist Rules 03 Class names can not contain special ...
Artist Rules 04 A class can not inherit from a leaf class
Artist Rules 05 Multiple Inheritance is not allowed.
Artist Rules 06 Attribute names can not contain spaces
Backup Options
Backup package before it will be modified by autocorrection
Use Baselines
Use XMI-Export
Export Path:
C:\Users\kwieland\AppData\Local\LieberLieber\EnArValidationRules
Disable Backup
Ok
Cancel
Help

Figure 14 Activating and deactivating validation rules in the C# code generator

In this dialog, you can also activate and deactivate auto-correction for the validation rules. In addition, you can also specify the backup options. Before the tool is modifying the model by auto-correction it exports the model as so-called baseline or as XMI file. The validation itself can be started by right-clicking a package in the model and by selecting “Extensions -> Artist -> Model Validation -> Start”.

The detected errors are listed in the “Model Validation Results” window (Figure 15).

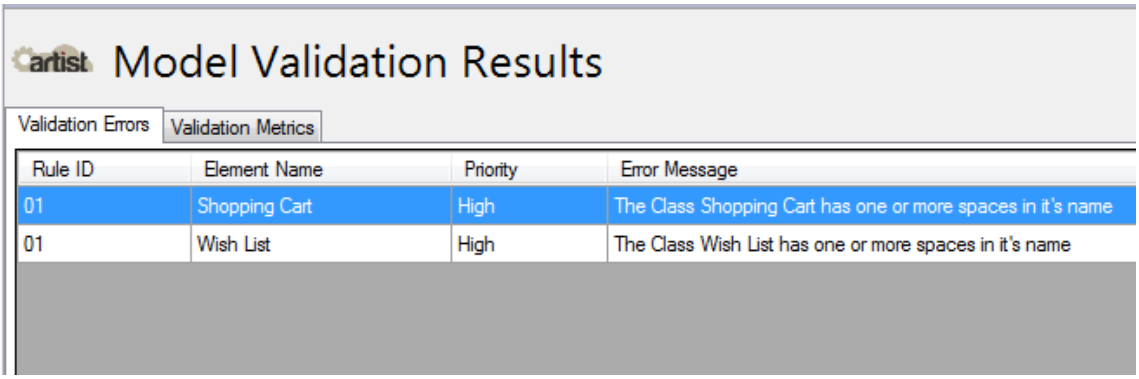
artist
Model Validation Results
Validation Errors
Validation Metrics
Rule ID
Element Name
Priority
Error Message
01 Shopping Cart High The Class Shopping Cart has one or more spaces in it's name
01 Wish List High The Class Wish List has one or more spaces in it's name

Figure 15 Detection of errors in the model validation results

In addition, some basic metrics are provided. Rules can be weighted according to is influencing the final result (Figure 16).

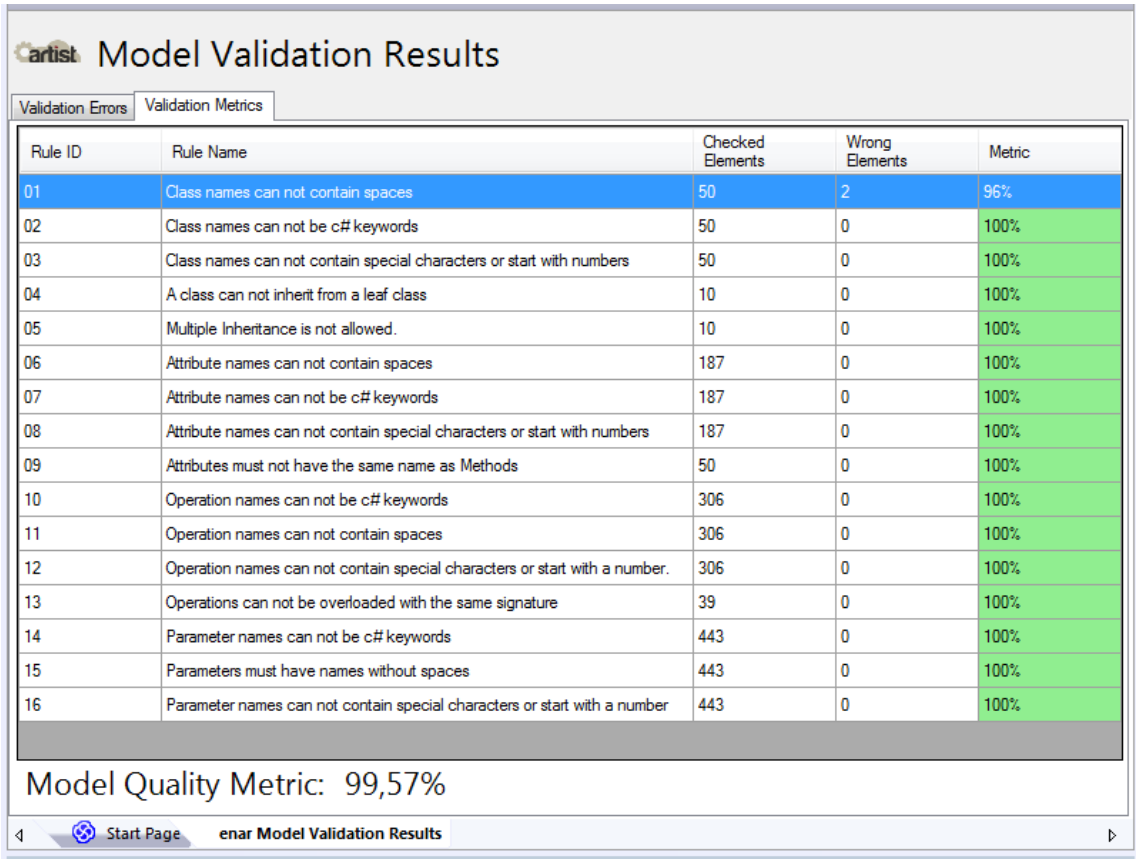
Model Validation Results
Validation Errors
Validation Metrics
Rule ID | Rule Name | Checked Elements | Wrong Elements | Metric
01 | Class names can not contain spaces | 50 | 2 | 96%
02 | Class names can not be c# keywords | 50 | 0 | 100%
03 | Class names can not contain special characters or start with numbers | 50 | 0 | 100%
04 | A class can not inherit from a leaf class | 10 | 0 | 100%
05 | Multiple Inheritance is not allowed. | 10 | 0 | 100%
06 | Attribute names can not contain spaces | 187 | 0 | 100%
07 | Attribute names can not be c# keywords | 187 | 0 | 100%
08 | Attribute names can not contain special characters or start with numbers | 187 | 0 | 100%
09 | Attributes must not have the same name as Methods | 50 | 0 | 100%
10 | Operation names can not be c# keywords | 306 | 0 | 100%
11 | Operation names can not contain spaces | 306 | 0 | 100%
12 | Operation names can not contain special characters or start with a number. | 306 | 0 | 100%
13 | Operations can not be overloaded with the same signature | 39 | 0 | 100%
14 | Parameter names can not be c# keywords | 443 | 0 | 100%
15 | Parameters must have names without spaces | 443 | 0 | 100%
16 | Parameter names can not contain special characters or start with a number | 443 | 0 | 100%
Model Quality Metric: 99,57%
Start Page
enar Model Validation Results

Figure 16 Metrics provided in the model validation results

After correcting the model by hand or by auto-correction, the code for the application can be generated. Several general and language specific parameters can be set as described in the Section **¡Error! No se encuentra el origen de la referencia.** of this document.

When right-clicking a package in the project browser and select “Code Engineering -> Generate Source Code” the dialog shown in Figure 17 is opened.

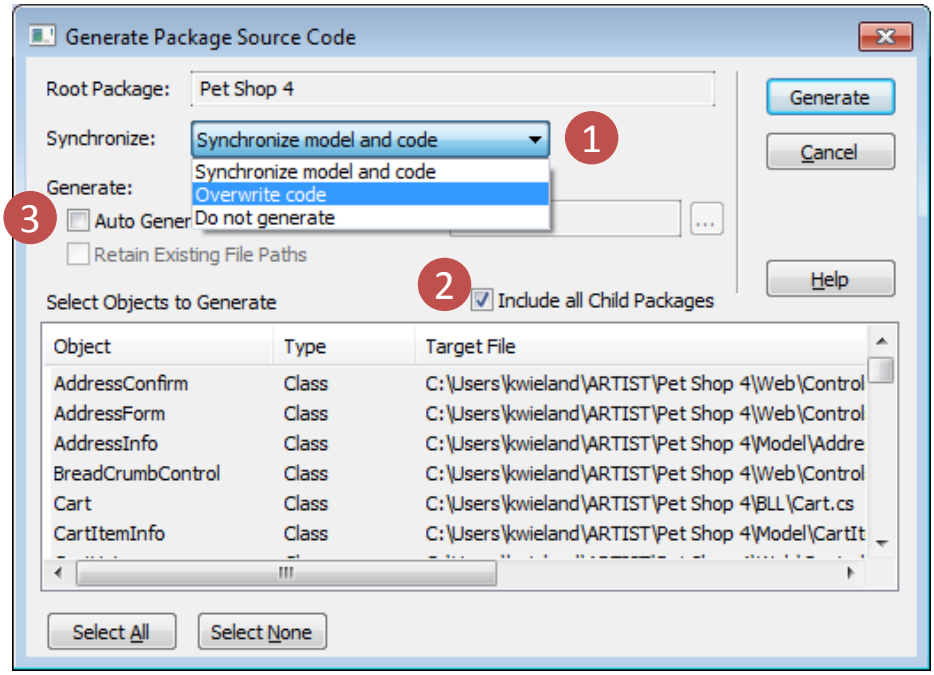

Figure 17 Dialog to generate code

Here, there are basically three different options:

1. If you want to follow a Roundtrip Engineering approach, that is to synchronize the model and code, both are updated accordingly to your changes. You can also select “Overwrite code”. With this option you override the before generated code.
2. If your model is structured with the help of packages, the “Include all Child Packages” checkbox should be selected for recursively looping all elements in the model for which code is generated.
3. Finally, if you want to specify a specific folder in which the code is generated, you have to select the “Auto Generate Files” checkbox. If activated you are forced to select a folder in your file system.

By clicking on “Generate” the source code is generated for each package, class, interface or enumeration including their properties, attributes and operations.

For each package in the model, an equally named folder is generated. You can change the namespace root by right-clicking a package in the project browser and by selecting “Code Engineering -> Set Namespace Root”.

After generating code, you can view the source code inside EA by selecting an element e.g. class, in the project browser and click on “F12”. The code editor including syntax highlighting is opening.

As described above, the source code templates can be adopted following your specific needs by clicking on “Tools -> Source Code Generation Templates…”.

4 Evaluation

The goal of this section is to describe our efforts of achieving the functional aspects (cf. Section 4.1) as well as the non-functional aspects (cf. Section 4.2) of the described code generators in Section 3. Overall, we aim for validation requirements for the code generators by establishing hard criteria which can be used for validation purposes. In particular, we are interested in the completeness and the correctness of the generated code with respect to the UML models as well as the maintainability and understandability of the generated code. For the first part, we employ a dedicated testing framework which is derived from the UML metamodel in order to guarantee that no concepts are missing as well as from formal semantic descriptions for UML. For the second part, we employ a formal metric framework which can be used for the generated code.

4.1 Completeness and Correctness of Generated Code

4.1.1 Completeness

For checking the completeness of code generators, there are several dimensions which contribute to the notion of completeness if we consider the term in general. First, completeness may refer to the model level where each input model should be transformable to an output. Second, completeness may refer to the metamodel level where each concept of the input metamodel may be reflected in the code generation. In the latter, we elaborate on the second meaning of completeness for transformations that consider the UML class diagrams as input and object-oriented programs as output.

In the appendix A of this document, we have documented the UML CD concepts based on the metamodel slicer capabilities reported in [15]. In particular, we used the following methodology to find the list of modeling elements of the UML CD:

1) Consider all concrete classes of the UML classes package
2) Use the metamodel slicer to push-down all inherited features to these concrete classes
3) Slice the concrete classes into a metamodel slice
4) Use a model to text transformation to produce a spreadsheet representation of the metamodel elements
5) Evaluate which concepts and features correspond to the programming level and which ones are currently not representable on the code level due to the impedance mismatch between UML and object-oriented programming languages (Java and C#)
6) Evaluate which remaining concepts and features are currently considered by code generators by running test executions and manually inspecting the code outcome.

Benefits of documenting the completeness of code generators are:

1) Contracts can be defined to check which models are valid or partially invalid input.
2) Code generators can be incrementally extended.
3) The impedance mismatch between UML and object-oriented programming languages (OOPLs) is more clearly characterized by documenting which elements of UML are challenging to represent on the OOPL level.

4.1.2 Correctness

For checking the correctness of code generators, we count on a set of test cases developed in UML. By comparing the expected output of translating them into code with respect to the actual output produced by the generators, we can check how correct the generated code is. A main concern in code generators is how to translate associations into code. Therefore, we

firstly focus here on the correction of translating associations as well as some of their attributes. As another example to demonstrate the complexity of the code generation capabilities, we discuss the generation of Enumerations in Java from UML Enumerations. While at a first glace this translations seems straight-forward, the possibility of defining inheritance relationships between UML Enumerations which is not supported on the Java level makes also this mapping challenging to implement.

Associations

Figure 18(a) shows the simplest case of a unidirectional association. The translation to an object-oriented language is quite straightforward. Thereby, the setter and getter methods for class A are very simple:

```
Class A{
   private B b = null;
   public B getB() {return b;}
   public void setB (B b) {this.b = b;}
}
```

Figure 18(b) shows the case where *b* must always have a value, i.e., it can never be null. Consequently, when creating an object of type A, the code generator has to force either the creation of a new instance of type B or associate A with any object of type B.

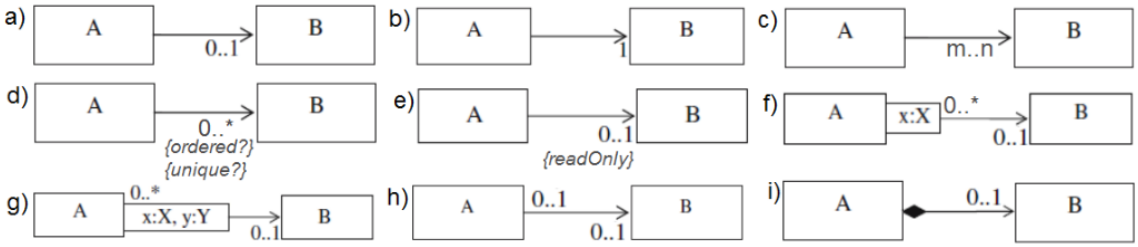

Figure 18 – Unidirectional Associations

Figure 18(c) displays the situation in which the target end of the association has lower and upper bounds established. In Java these checks must be done dynamically at runtime. That means, objects of type A can never reference less than *m* elements of type B or more than *n* of them. The use of exceptions in the setter method is the proposed mechanism to cope with this. Furthermore, this test case, as well as the one shown in Figure 18(d), exemplifies the use of collections in UML. There are four types of collections in UML, depending on the value of the *ordered* and *unique* properties of the association. Thus, if the association is ordered, we have a *Sequence* if it is not unique and an *OrderedSet* if it is unique. If the association is not ordered, we have a *Bag* if it is not unique and a *Set* if it is unique. The UML types *Set, Sequence* and *Bag* map straightforwardly with the Java types *java.util.Set*, *java.util.List* and *java.util.Collection*, respectively. However, there is no standard Java class that is equivalent to *OrderedSet.* Consequently, an option is to use a *HashSet* structure.

Unidirectional associations can become much more complex if association properties are used. For instance, another test case considers that the target end of the association is *readOnly* (Figure 18(e)). This means that the initial value associated to the reference object, *b*, can never be overwritten. An option could be to add an exception in the setter method checking if *b* has already been set (i.e., it is not null). However, a preferable solution is not to provide a setter method, so *b* has to be initialized elsewhere.

The test case in Figure 18(f) shows the use of qualifiers. By using qualifiers and in the simplest case – multiplicity 1 or 0..1, each element of type A may be associated with an element of type B by means of a qualifier value *x* (of type X). This means that each element A is connected to a set of tuples with type *TUPLE<x:X, b:B>*; and there is optionally a single element B for each possible element X. To better understand this, the authors in [2] propose the representation shown in Figure 19. For properties with single qualifying values, a solution is the following implementation (from [2]):

```
class A{
   Map<X, B> b = new HashMap<X, B>();
   public Collection<B> getB() { return b.values(); }
   public B getsB(X x) { return b.get(x); }
}
```

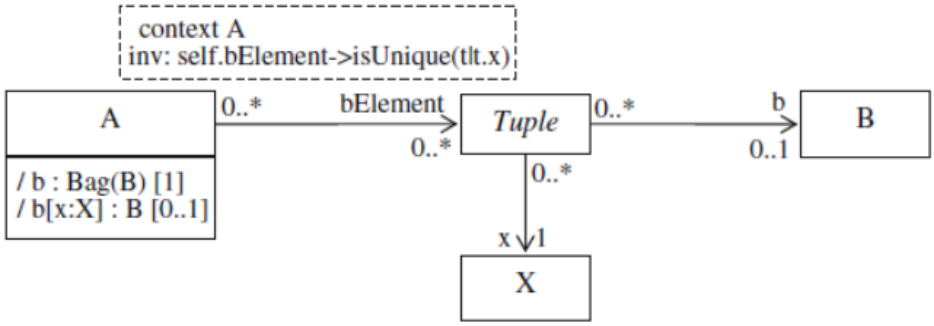

Figure 19 – Representation of qualifiers

When we have more than one qualifying value, as shown in Figure 18(g), the previous solution is not valid. Instead, we need a Map type on tuples, since now, starting from an element of type A, it is a tuple (of type <X,Y>) and not a single element that brings us to an element of type B. However, there is no direct equivalent of a TUPLE type in Java. Consequently, the code generator must create one when translating this kind of qualifiers. They need a proper way to map OCL tuples into Java. That means the tuple elements as well as the values for their fields are to be defined at the same time. Then a comparison of tuple elements should compare the names and values of the fields, which must be public.

Code generators also have to take into account the situation in which the source end multiplicity is not the default one (0..*). Thus, a 0..1 at the source end of the association, as shown in Figure 18(h), implies that only one element A may reference any single element B. However, there is no such primitive, for instance, in Java. The most general case is shown in Figure 20. The problem is that object oriented programming languages do not have any natural mechanism that equates to the *allInstances* operation of OCL. There is currently no mechanism for correctly translating these semantics into an object oriented programming language. Some proposals for enforcing the multiplicity [2] are disallowing uni-directional associations, implementing uni-directional associations as bi-directional and providing an implementation of the *allInstances* operation for all classifiers. However, none of them provides a fully correct solution.

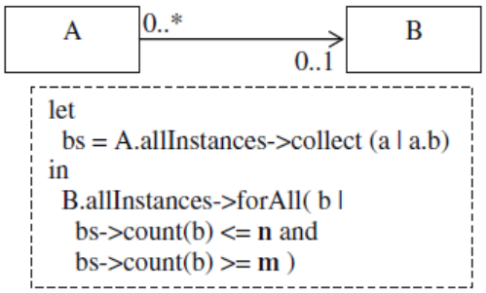

Figure 20 – Source end multiplicity

The situation regarding code generation for aggregation and composition associations (such as the one shown in Figure 18(i)) is still quite blurry. The modeling facilities of the current tools allow the violation of UML semantics such as having a part included in multiple composites at a time [5]. Indeed, a composite cannot be implemented by the standard association implementation of current object oriented programming languages.

Figure 21 – Bi-directional associations

In bi-directional associations, both ends are navigable. A one-to-one bi-directional association, as the one shown in Figure 21(b), can be considered as two opposing many-to-one associations with constraints ensuring that the bi-directional semantics are enforced, as displayed on Figure 22. Thus, in order to preserve the semantics of the bi-directional association, navigating from an object along the association and back again must reach the element you started from. An implementation for this is the code shown below [2].

```
Class A {
   B b = null;
   public B getB() {return b;}
   public void SetB (B b) {
      B old = this.b;
      this.b = b;
      if (old!=null) {old.setA(null);}
      if (b!=null && b.getA()!=this) {b.setA(this);}
   }
}
```

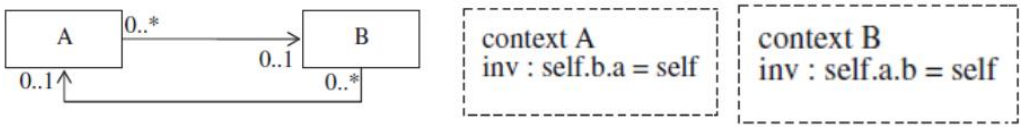

Figure 22 – Representation of bi-directional association

In bi-directional associations where one or the other end is a collection, as shown in Figure 21(b), code generators must add the code to the collection setters defined in the initial value for the collection based property, instead of adding the code for setting and resetting the other end of the property setter. Furthermore, as mentioned before with the different collection types in UML/OCL, code generators also need to take this into account. The

implementation of iterators in object-oriented languages is useful when we need to traverse collections.

Bi-directional qualified associations, as shown in Figure 21(c), are quite complex to implement. In the figure, the property B, navigable to from elements of type A, is qualified with a value of type X. Normally, a qualifier is added to a property with a "many" multiplicity in order to partition the set of elements, i.e., as if we were providing an index for all the elements in the collection. Thereby, in the figure there are a set of elements of type B associated to an element A.

Enumerations

As an example for Enumerations in UML, please confer the Figure 23. Enumerations have similar capabilities in UML as Classes (both inherit many powerful modeling capabilities from Classifier). Among these capabilities is the definition of inheritance relationships between Enumerations. For instance, one may define an Enumeration WeekDays that is extended by an Enumeration AllDays just providing the missing two days of the week. On the UML level, one now expects that for attributes typed with WeekDays, only the five possible values are valid, while for attributes typed with AllDays seven possible values are available, because AllDays inherits the literals of WeekDays.

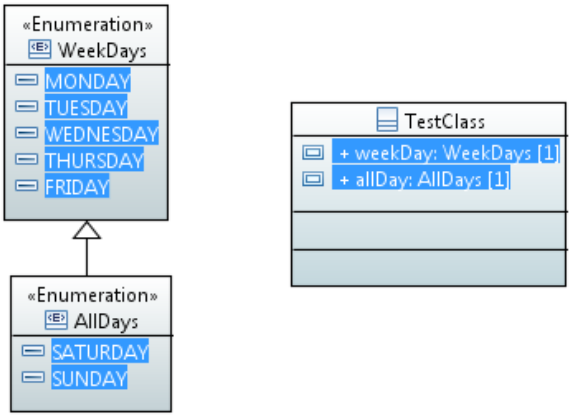

Figure 23 – Representation of inheriting Enumerations

In Java the situation is different, because Enumerations do not allow for inheritance (Enumerations are defined internally as Classes in Java which inherit from the base class Enumeration). In addition, there are also some major differences between inheritance between classes and enumerations. While all direct instances of a super class (instances of its sub classes) are valid instances of the super class, this is not transferable to enumerations and literals. Here only the literals defined for the super enumeration are valid literals for the super enumeration. The literals of the sub enumerations are not valid literals for the super enumeration. Thus, there is no upward type substitutability for super enumerations and sub enumerations as described before, only downward type substitutability (the literals of the super enumeration are valid literals for the sub literals).

In the following listing, we show how the example is compiled to Java code. Several workarounds are necessary on the Java code level. First, the literals of the super enumerations have to be duplicated in the sub enumerations. Second, specific converter functions have to be provided to cast the literals from top to bottom enumerations.

```
public enum WeekDays implements IDay{
```

```
    MONDAY("MONDAY"),
    TUESDAY("TUESDAY"),
    WEDNESDAY("WEDNESDAY"),
    THURSDAY("THURSDAY"),
    FRIDAY("FRIDAY");

    private final String description ;

    public String description() {
      return description ;
    }

    private WeekDays( final String description ) {
      this.description = description ;
    }

    public AllDays toAllDay(){
       return Enum.valueOf(AllDays.class, this.description());
    }
}

public enum AllDays implements IDay {
    MONDAY(WeekDays.MONDAY),
    TUESDAY(WeekDays.TUESDAY),
    WEDNESDAY(WeekDays.WEDNESDAY),
    THURSDAY(WeekDays.THURSDAY),
    FRIDAY(WeekDays.FRIDAY),
    SATURDAY("SATURDAY"),
    SUNDAY("SUNDAY");

    private final String description ;

    public String description() {
      return description ;
    }

    private AllDays( final WeekDays baseKind ) {
        this.description = baseKind.description() ;
     }

    private AllDays( final String description ) {
      this.description = description ;
    }

    public WeekDays toWeekDay() throws IllegalArgumentException {
       return Enum.valueOf(WeekDays.class, this.description());
    }

}
```

By having this specific realization of enumerations, there are some possibilities to emulate inheritance between enumerations on the Java code level. A client code using these enumeration definitions may look like this. Please note that the conversion downwards the enumeration hierarchy is not problematic, but upwards may result in exceptions as is the case in this example listing.

```
public class EnumTest {

	@Test(expected=IllegalArgumentException.class)
	public void test() {
		Clazz clazz = new Clazz();
		clazz.weekDay = WeekDays.TUESDAY;
		clazz.day = AllDays.SATURDAY;

		clazz.day = clazz.weekDay.toAllDay();

		clazz.day = AllDays.SATURDAY;


		// should through exception, because Sunday is not in the list
		of week days
		clazz.weekDay = clazz.day.toWeekDay();
	}

}
```

Wrap up. There are still many open issues and the need of a unified approach on the translation from UML to Java, and there is no a single solution that gathers all the advances in this research field. Furthermore, the release of Java 8 may have a positive outcome in this line, since some of its new features could be used to represent UML features and concepts that are not straightforward with previous editions of Java. We will report our advances in this field in D9.7 due to M30.

4.1.3 User Manual

In this section we explain how we can execute the test cases we have defined, namely to generate Java code from the UML specifications.

The set of test cases we have developed considers all the situations mentioned in the previous subsections. If a code generator is able to correctly convert all of them into code, then it will be complete and correct with respect to the conversion of associations along with most of their attributes. There are a total of 13 test cases, as it can be seen in Figure 24.

The project with the test cases is located in the ARTIST-Tooling Github Repository, under the path *migration/modernization/*, in the folder *eu.artist.migration.modernization.uml2java.testcases*. One can apply the code generator to any of the UML diagrams. For instance, in Figure 24 we show how to apply the Acceleo (Obeo) code generator. As a prerequisite, we need to have the Acceleo plugin installed. Then, we need to right-click on a UML diagram and select *Run As -> Acceleo UML to Java Generation*. A new project is automatically created with the created java code.

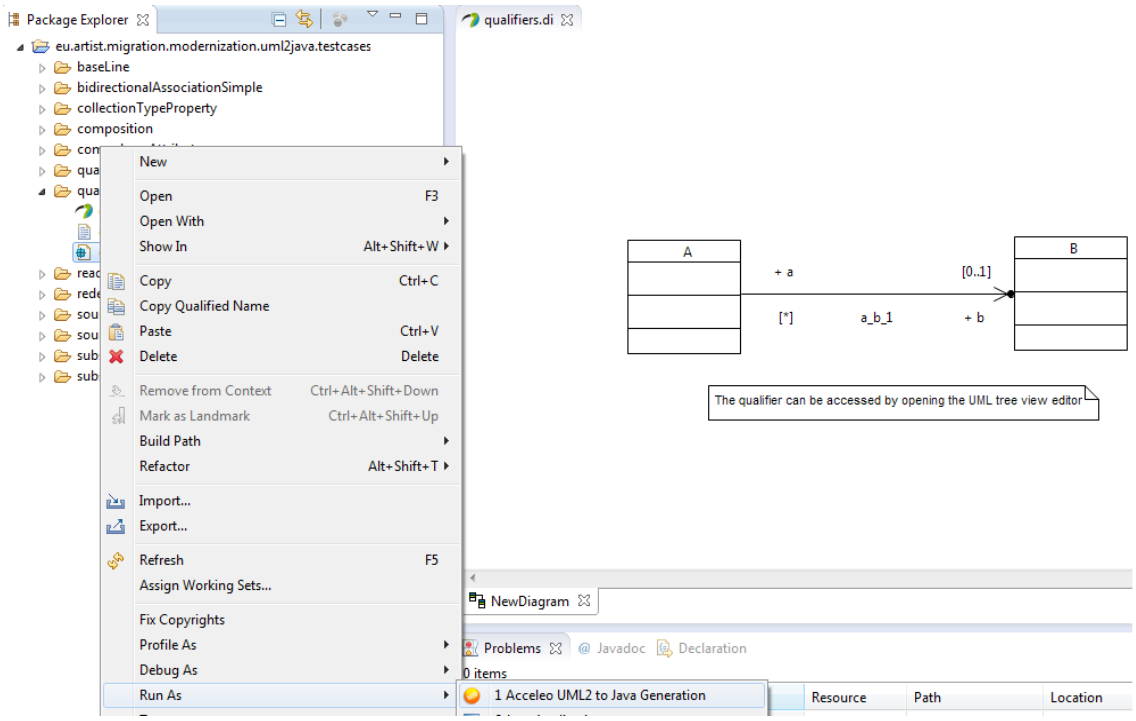

Figure 24 Executing the Acceleo (Obeo) code generator on a test case

4.2 Maintainability/Understandability of Generated Code

In the first stages of the ARTIST methodology (pre-migration phase) [17], [18], where the feasibility for a migration to the cloud is being evaluated, the maintainability metric (as defined by IEEE [19]) for calculating the software complexity is calculated. In the context of the modernization phase [20] the maintainability metric of the generated metric is calculated too. The main purpose for this re-calculation of the maintainability index is to analyse if within the application modernization process (through model to model and model to code transformations) this metric has improved.

Maintainability is defined in the IEEE standard glossary of Software Engineering [19] as "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment".

As explained in [18], in the context of ARTIST we use the compound MEMOOD method [20], to calculate the Maintenance based on the following model:

$$Maintenance = 2.399 + 0.493 \times Modifiability + 0.474 \times Understability + 0.524 \times Scalability + 0.507 * LOC$$

Where:

$$Modifiability = 0.629 + 0.471 \times NC - 0.173 \times NGen - 0.616 \times NAggH - 0.696 \times NGenH + 0.396 \times MaxDIT$$

$$Understability = 1.66 + 0.256 \times NC - 0.394 \times NGenH$$

$$Scalability = 0.182 \times 0.99 \times AC + 0.100 \times EC + 0.097 \times ND - 0.036 \times PC + 0.068 \times DMS$$

$$LOC = 0.269 + 0.008 \times Coupling + 0.181 \times cohesion + 0.119 \times CC + 0.084 \times ILCC$$

with NC: Number of classes; NGen: Number of Generalizations; NGenH: Number Of Generalizations Hierarchies; AC: Afferent Coupling; EC: Efferent Coupling; ND: Nesting Depth; DMS: Distance from main sequence; PC: % coverage; CC: Cyclomatic complexity; ILCC: IL Cyclometic complexity.

In the first version of the prototype some of the metrics are calculated (mainly those needed to calculate LOC and Scalability) for the generated code. In the next versions of the prototype, the other needed atomic metrics to calculate the maintainability will also be provided. The final objective is to calculate both metrics, the one obtained from the source code and the one calculated from the generated code, and analyse possible variations.

4.2.1 Technical description

4.2.1.1 Prototype architecture

The current Maintainability metric calculator prototype architecture is a java API that explores source files to generate several Metrics of a specific project. Figure 25 depicts the overall architecture:

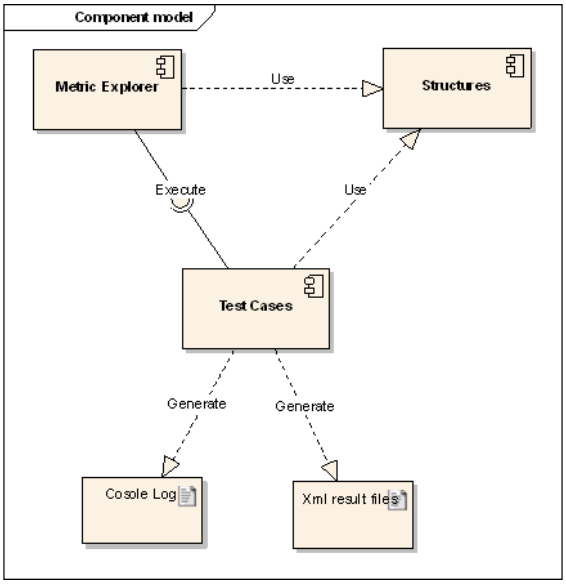

Figure 25 – MMC High level architecture

While the main objective of the Maintainability Metric Calculator is to expose an API that any other plug-in or RCP could use to obtain the metrics generated in the Artist project, it also provides Test Cases to access the same functionality as if used programmatically. The generated metrics is available in XML files and console log.

4.2.1.2 Components description

The current Maintainability Metric calculator prototype component comprises three components:

- **Metric Explorer:** This is the main component of the Maintainability Metric calculator current prototype. It provides the calculation of all the required metrics that are used

to generate the maintainability metric. Besides, it also provides exporting features to convenient formats like XML or JSON**.**

- **Structures:** This component contains the structures of the inputs and outputs models that the Metric Explorer uses. It also provides the functionality for generating the output file formats (XML, JSON).
- **Test Cases:** This component is provided for implementing the testing of the Metric Explorer component. It generates several use cases that test the functionality of the Maintainability Metric calculator. The test case generates console logs and XML files with several examples (DEWS and JavaPetStore).

4.2.1.3 Technical specifications

All the components are developed in JavaSE 1.6. So this is the minimum java version for executing the API. There are no any other requirements.

User Interface

There is no user interface implemented as the result obtained from Maintainability Metric calculator will be consumed by other ARTIST tools or used for validation. For executing the Metrics Explorer API the user has to execute the test cases included in the API. Several input parameters can be changed for obtaining new metrics of different projects.

Back-end

There is no persistency implemented neither planned for this API. Thus, every time the new metrics are needed, they have to be calculated. Persistency is delegated to API consumers.

4.2.2 Delivery and usage

4.2.2.1 Package information

Figure 26 depicts the package structure of the main component, the Artist metrics generator plug-in.

- **eu.artist.migration.mmc.metricexplorer**: Contains the classes for exploring the UML models and the source code.
- **eu.artist.migration.mmc.strucctures**: Contains the classes of the structures used by the metric explorer component
- **eu.artist. migration.mmc.test**: Contains the test cases classes for executing the metric explorer component.

4.2.2.2 Installation instructions

In this version of the prototype this plug-in requires manual installation. The user has to import de components to the Eclipse workspace manually.

Requirements. All the components are developed in JavaSE 1.6. So this is the minimum java version for executing the API.

4.2.2.3 User Manual

Import the project into the Eclipse workspace as in Figure 27.

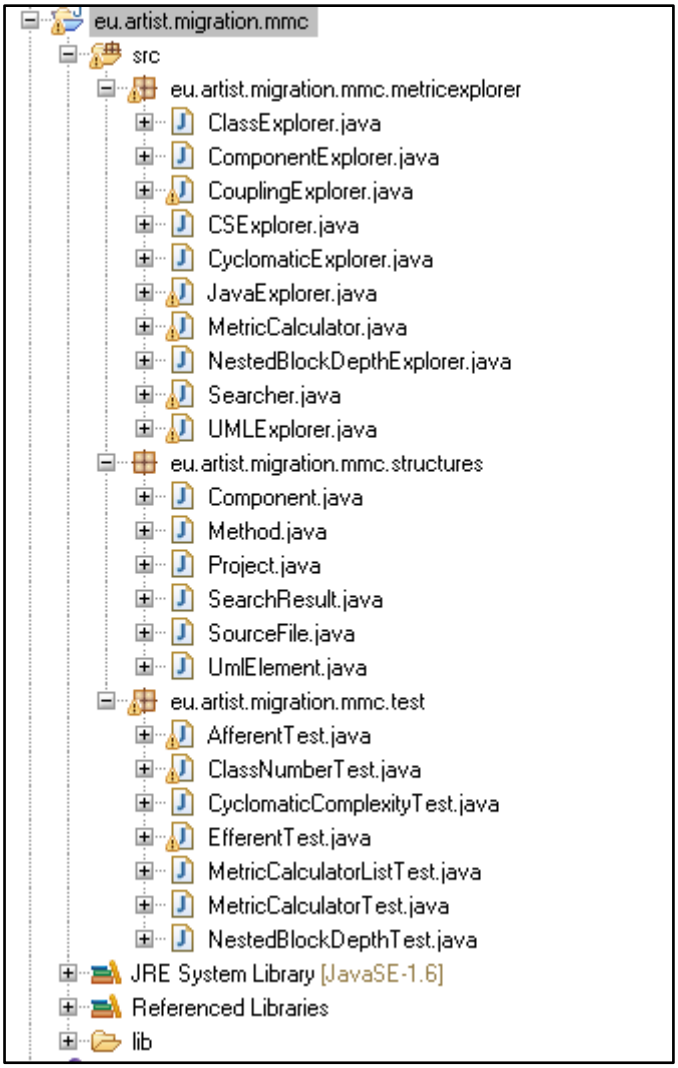

Figure 26 Package structure of the Maintainability metric calculator

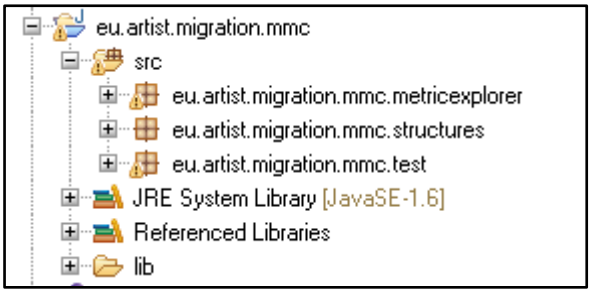

Figure 27 Maintainability metric calculator project

Open the eu.artist.migration.mmc.test package, as shown in Figure 28.

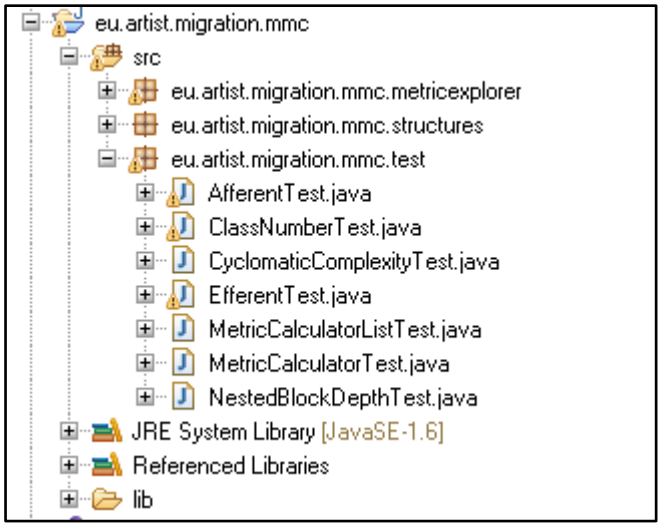

Figure 28 Testing package

Right click in a test case and select the Run as Java Application option (cf. Figure 29).

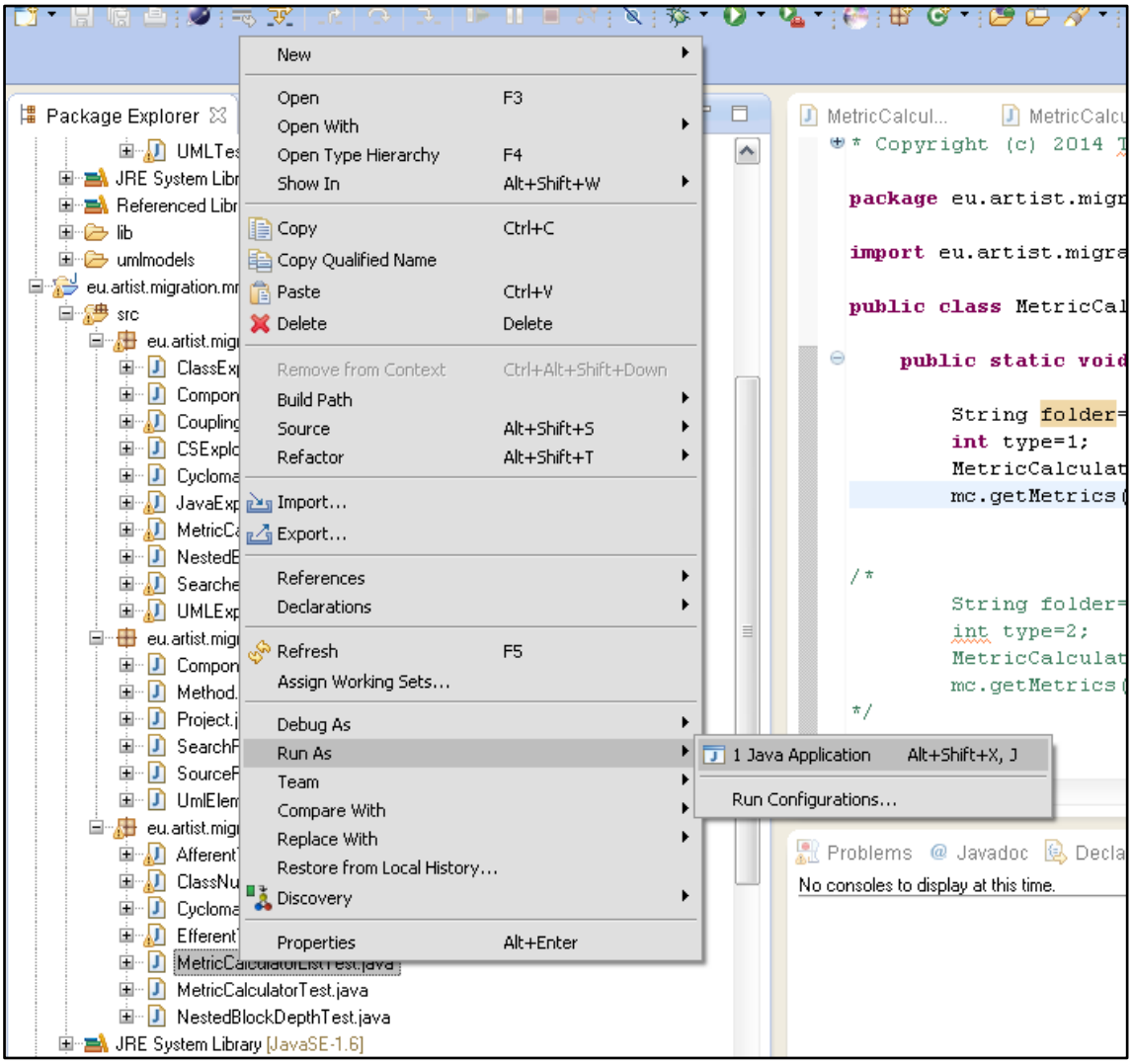

Figure 29 Maintainability metric calculator testing

***Note**: The user has to change the "hardcoded" input parameters of the test cases manually*

4.2.2.4 Licensing information

This component is offered under EPL license.

4.2.2.5 Download

This release of the Maintainability metric calculator is available in the ARTIST github, more precisely at the following address:

https://github.com/artist-project/ARTIST-Tooling/tree/master/migration/modernization/eu.artist.migration.modernization.mmc

5 Delivery and Usage

5.1 Package Information

In this deliverable, the described components have been developed either for the Eclipse environment or the Enterprise Architect. In case of Eclipse, the components are packaged either as Eclipse plugins or Eclipse projects.

5.2 Installation Instructions

Components that have been realized in terms of Eclipse plugins need to be located in the respective *Eclipse plugins* folder. These components provide a dedicated UI to execute them. How they can be used is describe in the respective user manuals.

Components that are not yet realized in terms of Eclipse plugins are provided as Eclipse projects. They need to be located in the respective Eclipse workspace to execute them. We have prepared dedicated configuration to launch them in Eclipse. Again, how they can be launched is described in the respective user manuals.

Components that have been realized for the Enterprise Architect are integrated into this environment.

5.3 User Manual

User manuals are provided by the sections describing the corresponding component.

5.4 Licensing Information

Generally, the objective is to propose the Model Understanding Toolbox as an open source solution resulting from the ARTIST project.

The selected license is the Eclipse Public License (EPL) [21] which is a known as a “commercial-friendly” open source license. This should facilitate the future potential reuse and integration of the toolbox (or at least of some of its components) by external partners.

5.5 Download Instructions

The sources of the different components newly developed during this second year of the ARTIST project have been grouped in a Zip file that has been put on the ARTIST (Livelink) repository at the following location:

https://dms-prext.fraunhofer.de/livelink/livelink.exe?func=ll&objId=4110632&objAction=browse&viewType=1

6 Conclusions

In this deliverable we have documented the current state of the ARTIST code generation capabilities as well as the underlying design rationale of the established toolkit consisting of two generic UML2Java and UML2C# code generators implemented as M2T transformations, one residing in Eclipse based on Acceleo and the other one residing in Enterprise Architect based on CTF. We have also shown how the code generators are extensible for particular cloud platforms and discussed issues in current open-source and commercial code generators. We emphasize the importance to focus on completeness and correctness of the code generators as these aspects are as important as in the context of compilers. Thus, a general methodology for developing code generators is needed. We tackled these challenges by providing a minimal core of the UML Class Diagram language dedicated test cases which should be fulfilled by the code generators.

The next steps concerning the code generation toolkit is to provide further extensions of the base code generators for additional PaaS providers and dedicated APIs which are preferable to be used in cloud environments in the context of WP10. Furthermore, we plan to extend our validation framework with additional test cases for testing the extensions of the code generators. Finally, we aim to contribute our findings and code generation capabilities to a new emerging Eclipse project “Eclipse UML Generators” (http://eclipse.org/umlgen) that aims for providing open-source implementations of UML-based code generators developed with Acceleo.

7 References

[1] ARTIST, «D9.3 - Migration Rules Formalized as Generic Model Transformations,» 2014.

[2] D. H. Akehurst, G. Howells and K. McDonald-Maier, "Implementing associations: UML 2.0 to Java 5," *Software & Systems Modeling,* vol. 6, no. 1, pp. 3-35, 2007.

[3] J. O. Blech, S. Glesner and J. Leitner, "Formal Verification of Java Code Generation from UML Models," in *Proceedings of the 3rd International Fujaba Days*, 2005.

[4] G. Engels, R. Hücking, S. Sauer and A. Wagner, "UML Collaboration Diagrams and Their Transformation to Java," in *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99)*, 1999.

[5] D. Gessenharter, "Mapping the UML2 Semantics of Associations to a Java Code Generation Model," in *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, 2008.

[6] Y.-G. Guéhéneuc, "A Systematic Study of UML Class Diagram Constituents for their Abstract and Precise Recovery," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, 2004.

[7] G. Génova, C. Ruiz del Castillo and J. Llorens, "Mapping UML Associations into Java Code," *Journal of Object Technology (JOT),* vol. 2, no. 5, pp. 135-162, 2003.

[8] K. Hanada, K. Okano, S. Kusumoto and K. Miyazawa, "Practical Application of a Translation Tool from UML/OCL to Java Skeleton with JML Annotation," in *Proceedings of the 14th International Conference on Enterprise Information Systems*, 2012.

[9] W. Harrison, C. Barton and M. Raghavachari, "Mapping UML Desings to Java," in *Proceedings of the 15 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, 2000.

[10] T. Sturm, J. von Voss and M. Boger, "Generating Code from UML with Velocity Templates," in *Proceedings of the 5th International Conference on Model Driven Engineering Languages and Systems (MoDELS'02)*, 2002.

[11] P. Sánchez, P. Letelier, J. A. Pastor and J. A. Ortega, "A Framework to Translate UML Class Generalization into Java Code," in *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, 2002.

[12] M. Usman and A. Nadeem, "Automatic Generation of Java Code from UML Diagrams using UJECTOR," *International Journal of Software Engineering and Its Applications (IJSEIA),* vol. 3, no. 2, pp. 21-38, 2009.

[13] ARTIST, «D7.3 - Analysis and incorporation of third party offerings,» 2014.

[14] ARTIST, «D8.2.2 - Components for Model Discovery from Legacy Technologies,» 2014.

[15] ARTIST, «D8.3.2 - Mechanisms for Viewpoint Definition and View Extraction from Models of Legacy Artifacts,» 2014.

[16] ARTIST, «D9.7 - Integrated environment for maintaining/developing forward engineering process,» 2015.

[17] ARTIST, «ARTIST Methodology M24,» 2014.

[18] ARTIST, "D5.1.1 - Specification of the Business and Technical Modernization Assessment in ARTIST," 2013.

[19] IEEE, "IEEE Standard Glossary of Software engineering Terminology," 1990.

[20] S. Muthanna, K. Kontogiannis, K. Pannambalam and B. Stacey, "A maintainability model for industrial software systems using desing level metrics," in *Proceedings of the IEEE 7th Working Conference on Reverse Engineering*, 2000.

[21] "Eclipse Public License," [Online]. Available: http://www.eclipse.org/legal/epl-v10.html.

APPENDIX: Core elements of the UML Class Diagram Language

MetaClass	MetaProperty
Association	
	attribute
	elementImport
	feature
	general
	generalization
	importedMember
	inheritedMember
	isAbstract
	isDerived
	isFinalSpecialization
	isLeaf
	member
	memberEnd
	name
	nameExpression
	namespace
	navigableOwnedEnd
	ownedComment
	ownedEnd
	ownedMember
	owner
	package
	packageImport
	powertypeExtent
	qualifiedName
	redefinedClassifier
	redefinedElement
	redefinitionContext
	visibility
Class	
	attribute
	classifierBehavior
	elementImport
	feature
	general
	generalization
	importedMember
	inheritedMember
	interfaceRealization
	isAbstract
	isActive

	isFinalSpecialization isLeaf member name nameExpression namespace nestedClassifier ownedAttribute ownedBehavior ownedComment ownedConnector ownedMember ownedOperation owner package packageImport powertypeExtent qualifiedName redefinedClassifier redefinedElement redefinitionContext role visibility
Comment	 annotatedElement body ownedComment owner
ElementImport	 alias importedElement importingNamespace ownedComment owner source target visibility
Enumeration	 attribute elementImport feature general generalization importedMember inheritedMember

	isAbstract isFinalSpecialization isLeaf member name nameExpression namespace ownedAttribute ownedComment ownedLiteral ownedMember ownedOperation owner package packageImport powertypeExtent qualifiedName redefinedClassifier redefinedElement redefinitionContext representation visibility
EnumerationLiteral	 enumeration name nameExpression namespace ownedComment owner qualifiedName slot visibility
Generalization	 general generalizationSet isSubstitutable ownedComment owner source specific target
GeneralizationSet	 generalization isCovering isDisjoint

	name nameExpression namespace ownedComment owner powertype qualifiedName visibility
Interface	attribute elementImport feature general generalization importedMember inheritedMember isAbstract isFinalSpecialization isLeaf member name nameExpression namespace nestedClassifier ownedAttribute ownedComment ownedMember ownedOperation owner package packageImport powertypeExtent qualifiedName redefinedClassifier redefinedElement redefinedInterface redefinitionContext visibility
InterfaceRealization	contract implementingClassifier mapping name nameExpression namespace

	ownedComment owner qualifiedName source target visibility
Model	 elementImport importedMember member name nameExpression namespace ownedComment ownedElement ownedMember owner packageImport qualifiedName viewpoint visibility
Operation	 bodyCondition class concurrency datatype elementImport featuringClassifier importedMember interface isAbstract isLeaf isQuery isStatic member method name nameExpression namespace ownedComment ownedMember ownedParameter ownedParameterSet owner packageImport

	postcondition precondition qualifiedName raisedException redefinedElement redefinedOperation redefinitionContext visibility
Package	 URI elementImport importedMember member name nameExpression namespace nestingPackage ownedComment ownedElement ownedMember owner packagedElement packageImport qualifiedName visibility
PackageImport	 importedPackage importingNamespace ownedComment owner source target visibility
Parameter	 defaultValue direction effect end isException isOrdered isStream isUnique lower lowerValue name

	nameExpression namespace operation ownedComment owner parameterSet qualifiedName type upper upperValue visibility
PrimitiveType	 attribute elementImport feature general generalization importedMember inheritedMember isAbstract isFinalSpecialization isLeaf member name nameExpression namespace ownedAttribute ownedComment ownedMember ownedOperation owner package packageImport powertypeExtent qualifiedName redefinedClassifier redefinedElement redefinitionContext visibility
Property	 aggregation association associationEnd class datatype

	defaultValue end featuringClassifier interface isDerived isDerivedUnion isID isLeaf isOrdered isReadOnly isStatic isUnique lower lowerValue name nameExpression namespace ownedComment owner owningAssociation qualifiedName qualifier redefinedElement redefinedProperty redefinitionContext subsettedProperty type upper upperValue visibility